COMPILATION OF PARALLEL APPLICATIONS VIA AUTOMATED TRANSFORMATION OF BMF PROGRAMS

By
Brad Alexander
March 28, 2006

A THESIS SUBMITTED FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
IN THE SCHOOL OF COMPUTER SCIENCE
UNIVERSITY OF ADELAIDE

March 28, 2006

Preface

Transformation is crucial to any program improvement process. Highly transformable notations pave the way for the application of deep and pervasive program improvement techniques. Functional programming languages are more amenable to transformation than their more traditional imperative counterparts. Moreover, functional programs specify only true dependencies between values, making improvements that reveal and exploit parallelism much easier. Some functional programming notations are more transformable than others. Bird-Meertens-Formalism (BMF) is a functional notation that evolved as a medium for transformational program development. A substantial, and growing, body of work has created novel tools and techniques for the development of both sequential and parallel applications in BMF.

Formal program development is at its most useful when it can be carried out automatically. Point-Free BMF, where programs are expressed purely as functions glued together with higher-order operators, provides enhanced scope for automated development because many useful transformations can be expressed as easily applied re-write rules. Moreover, realistic sequential and parallel static cost models can be attached to BMF code so the relative merits of applying various transformations can be accurately assessed.

In spite of its potential merits there has been little work that has utilised point-free BMF, in a pervasive manner, as a medium for automated program improvement. This report describes a prototype implementation that maps a simple point-wise functional language into point-free BMF which is then optimised and parallelised by the automated application of, mostly simple, rewrite rules in a fine-grained and systematic manner. The implementation is shown to be successful in improving the efficiency of BMF code and extracting speedup in a parallel context. The report provides details of the techniques applied to the problem and shows, by experiment

and analysis, how reductions in high data-transport costs are achieved. We also describe techniques used to keep the optimisation task tractable by alleviating the hazard of case-explosion.

The report is structured according to the stages of the compilation process, with related work described at the end of each chapter. We conclude with our main finding, namely, the demonstrated feasibility and effectiveness of optimisation and parallelisation of BMF programs via the automated application of transformation rules. We also restate techniques useful in achieving this end, the most important of which is the substantial use of normalisation during the optimisation process to prepare code for the application of desirable transformations. We also present a brief summary of potential future work including the introduction of more formally described interfaces to some of the transformative rule-sets, the automatic production of annotated proofs and a facility to display static estimates of the efficiency code during transformation.

Declaration

This work contains no material which has been accepted for the award of any other degree or diploma in any university or tertiary institution and, to the best of my knowledge and belief, contains no material previously published or written by another person, except where due reference has been made in the text. I give consent to this copy of my thesis, when deposited in the University Library, being available for loan and photocopying.

Bradley Alexander
March 28, 2006

Acknowledgments

I thank my supervisor Andrew Wendelborn for his unwavering dedication in reading the many long drafts of this work (any remaining errors are entirely my own) and for his wise counsel during the course of this project.

I also thank all of my colleagues, every one of whom is encouraging and supportive. In particular, I am grateful to Paul Roe, for his initial ideas for the Adl language, Dean Engelhardt, for his long collaboration on this project, all of my past honours and masters students, of whom Ingrid Ahmer, Paul Martinaitis, Greg Peterson, Joseph Windows, Dean Philp are major contributors to this project. I am also grateful to the team at INRIA who built and maintained Centaur. In particular, I thank Thierry Despeyroux for his prompt and cheerful answers to my naïve technical questions.

Finally, I am extremely grateful to my family and my wife's family for all their moral support and encouragement and, most of all, I thank my wife Katy, and my children, Thomas and Rebecca for their love, support and patience.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Distributed Parallel Computing

Parallel computing has much potential. Parallel computers have substantially more computing power and memory than their sequential contemporaries. Current distributed parallel computers, in particular, offer scalable high-performance computing, often, at not-unreasonable cost.

However, distributed parallel computing is hard. Computing tasks need to be coordinated to contribute to a shared solution without interfering with each other. Large, and increasing, volumes of data need to be managed carefully to minimise communications costs. These factors mean program construction, debugging, tuning and maintenance is more difficult in a parallel environment than in a sequential environment. Errors in parallel programs are often manifest in the form of unexpectedly bad performance or intermittent incorrect results, deadlocking, or program crashes. These problems are not helped by the lack of a universally accepted high-level programming model for parallel computing. As a consequence much programming happens at a low level where many difficult details of distributed parallel computing are exposed to the applications programmer. When a low level program has to be ported to a new architecture, existing parallel programs have to be manually re-tuned or re-written to extract more performance and/or just work correctly. This costly migration process adds a large premium to the cost of owning parallel software. As a result, distributed parallel computing is confined mainly to small but nonetheless important niche application areas such as, geophysics, computational fluid dynamics, computational chemistry/molecular-biology, particle and gravitational physics, finite element methods, large scale search algorithms, image-processing, and data mining. Such applications attract large enough returns to justify substantial investments in parallel software. That is, distributed parallel computing is primarily used by people who really want the extra performance and are prepared to pay for it. This small market does not produce the economies of scale required to drive the development of hardware and software tools at the same pace as mainstream computing thus maintaining disincentives to the widespread use of distributed parallel computing.

This problem would be alleviated by a widely-accepted high-level model of parallel computing. Such a model needs to be:

General purpose: easily applicable to most applications,

Simple: programs easily written, understood, and verified,

Efficient: exhibiting predictable performance with a low performance penalty for using the model rather than some low-level programming notation, and

Portable: applicable to a wide range of architecture without major changes to programs.

To date, there is no parallel distributed model that is widely believed to meet all of these criteria[1] so there is still, much, scope for work to be done.

1.2 Implementing a model

It is the task of a compiler and runtime system to efficiently bridge the distance between the programming model and the underlying architecture. The gap between a sufficiently simple programming model and a distributed architecture is very large [2]. In order to make the construction of the compiler/runtime system feasible implementors typically make one or more of the following compromises.

Making parallelism explicit in the source language: Leads to a relatively simple implementation but leaves most of the difficult work to the programmer.

[1]See the survey article by Skillicorn and Talia[130] comparing a wide range of parallel models.

[2]As with many things, an exact definition of *sufficiently simple* depends very much on the audience. A seasoned parallel applications programmer may find C or Fortran-MPI is sufficiently simple, leading to a small gap between programming model and machine. However, we believe most programmers would prefer a programming model where the details of parallel implementation are less explicit.

Restricting the source language: Restricting the programmer to common patterns of computation or data-access in order to limit the number of cases the compiler has to handle and also the scope for introducing inefficiency.

Compromising efficiency: Adding a layer of interpretation that results in a loss of efficiency for some programs.

The, current, most-widely used programming languages/libraries for distributed platforms make the first compromise, examples include MPI[105] and PVM[51].

Many models make the second compromise by restricting the ways in which parallel programs can be expressed. Data parallel languages such as Data-Parallel C-Extensions (DPCE)[106], Connection-Machine Lisp[132] and High Performance Fortran (HPF)[118] allow a single thread of control but admit parallelism through bulk operations on aggregate data. Skeletons support the same sort of bulk operations but extend these with more general templates for composing parallel programs[3]. Examples of Skeleton implementations include P3L[31] and SCL[43]. Other restricted notations include LACS[116] and ZPL[107] which require the user to express parallel programs in terms of transformations to arrays. As a general rule, these models with restrictions are, at the cost of some flexibility, much easier for programming than more explicit models.

Implementations making the third compromise offer a less-restricted programming model at the cost of a layer of interpretation that can be costly to maintain at runtime. Implementations making this compromise include abstractions for DSM[69] and distributed graph reduction[135]. The extent to which performance is compromised is heavily dependent on the sophistication of the implementation and the characteristics of the applications run on it.

As a final note, it should be said that some implementations may not fit neatly into the taxonomy above. Significant examples include various implementations of the BSP model[137, 76] which achieves good performance by restricting itself to applications expressed as a sequence of composed supersteps, and preventing the programmer from mapping tasks to physical processors. However, the model requires the programmer to to explicitly specify, the tasks, and the communications that are to take place between them.

[3]At their most general, Skeletons can be as explicit as more general models. An example is the `DMPA` *Dynamic-Message-Passing-Architecture* skeleton[42]. But such instances are relatively rare.

Some restricted implementations such as Fortran 90, HPF and Data-Parallel C-Extensions also require the programmer to specify alignment information and thus have a degree of explicitness. Finally, Vectorising Fortran compilers[10] (section 6.2.1, pp 305–309) provide a very implicit programming model, they extract parallelism from extant sequential code, but, in general, the effective extraction of parallelism requires a very sophisticated compiler.

This concludes our informal classification of models with implementations targeted to distributed-memory machines. This provides a context for the language implementation described in this report.

1.3 This work

This report describes an experimental implementation, targeted to a distributed parallel platforms, Our implementation is based on the source language Adl. Adl is small, strict, functional language where parallelism is expressed through bulk operations on arrays, or vectors, as they are called in this work. These bulk operations include the familiar data-parallel functions: `map`, `reduce` and `scan`. Adl allows these operations to be composed and tupled. Adl also provides constructs for selection, iteration and the introduction of new scopes. In addition Adl supports arbitrary random access to vector elements and other variables in scope and dynamic allocation of vectors. Chapter 2 provides a number of concrete examples of Adl programs.

The current version of Adl does not support recursive definitions of functions. Instead the language, and its implementation are directed toward the primary goal of expressing parallelism through bulk operators over aggregates. The recursion is likely to be included in future versions.

The parts of Adl The Adl project consists of two parts[7]. The earlier part, based on a highly optimised multithreaded model[54], supports efficient nested parallelism over irregular structures by generating many small, overlapping communications. The second part[8], the one described in this report, uses transformational programming to generate and optimise aggregated communications.

Implementation challenges Though Adl does restrict the way in which parallelism is expressed, compilation of Adl still presents substantial challenges,

```
sum_sqrs(a:vof int) :=
  let
     sqr x := x * x;
     add (x,y) := x + y
  in
     reduce(add,0,map(sqr,a))
  endlet
```

(a)

$$+/_0 \cdot (\times \cdot (\mathsf{id}, \mathsf{id})^\circ)*$$

(b)

Figure 1. Adl source for **sum_sqrs** (part (a)) and an equivalent point-free version (part (b)).

though parallelism is easily identified through data parallel operations such as map, the distribution and communication of data remains implicit. Furthermore, Adl places no restrictions on which values are available inside of its parallel operations. Any value in scope can be arbitrarily referenced from inside these operations and, in particular, arbitrary indexing into any vector value is allowed from within these operations. This places the onus on the implementor to generate, consolidate, and minimise the communication that is implicit in these references.

In order to bridge this gap between what is, essentially, the shared-memory model of Adl language and the distributed memory of the target architecture the implementation must, at some stage, convert some arbitrary references to values into explicit communication[4]. The implementation described in this report, henceforth described as "the Adl implementation", performs this conversion very early in the compilation process by translating Adl code into *point-free-form.* Programs written in point-free form[13, 59, 38] consist only of functions, no variables are allowed.

Figure 1 shows Adl source code for a program to calculate the sum of squares of an input vector (part (a)) and its point-free equivalent. The source program is reasonably conventional, using the **map** primitive to apply the **sqr** function to each element of the input vector and the **reduce** function to sum the list. The point-free version consists entirely of functions glued together using function composition,

[4]In the presence of arbitrary indexing into vector values, the final specification of some communications must, by necessity, be delayed until run-time. However, many communications can be specified statically and an efficient framework can be set up for the rest.

denoted: $\cdot$[5] , and the tupling construct: $(\ldots)^\circ$ which applies all of its constituent functions to its input value storing results in an output tuple. The $*$ symbol denotes a map function that applies the prefixed function to each element of an input vector. The / symbol denotes a **reduce** function that inserts the prefixed binary function between the elements of an input vector. The point-free version is completely free of variables for the runtime storage of values.

The lack of variables in point-free form means that functions are not only responsible for computation on values, but also responsible for determining the movement of values to functions to be executed in later parts of the program. In, other words, in point-free programs, must contain functions to route as well as compute data. Many of these *routing-functions* will still be present, in optimised and distributed form, in the final parallel code.

To summarise: once code is in point-free form, data movement is explicit; once data-movement is explicit, it can be optimised by replacing functions that generate a lot of data movement with functions that generate less data movement. Similarly, point-free code can be made explicitly parallel by inserting functions with parallel semantics into the code. Note that all of this processing can only take place if point-free form can:

1. express the required sequential and parallel functions

2. be transformed in the way required by this application

Fortunately, there is a large body of literature describing an algebra[6] and its notation, that together meet these requirements very well.

The Bird-Meertens formalism The Bird-Meertens Formalism (BMF) is a set of theories of types and operations over those types[17, 18, 21, 20, 131, 58, 15, 114]. BMF was developed as a medium for transformational programming. The idea is to start with an obviously correct but inefficient specification of a program and transform it, by the incremental application of correctness-preserving steps, to a much more efficient but less obvious version. Because transformation is much easier in a functional context

[5]We use $\cdot$ rather than the traditional $\circ$ to denote function composition to make code slightly more compact.

[6]Actually, a family of algebras for different data types.

than an imperative context[7], BMF programs have functional semantics. Most BMF programs work with aggregate types such as lists. The notation is often, but not always, point-free and quite terse[8], for example:

$$sqrt * \cdot sqr *$$

expresses a program to find the absolute values of an input array of numbers[9]. A large number of simple and broadly applicable program identities can be used to transform BMF programs. For example, the identity $f * \cdot g* = (f \cdot g)*$ can be used to fuse the two map functions in the above program into one:

$$\begin{array}{ll} & sqrt * \cdot sqr* \\ = & \qquad \{f * \cdot g* = (f \cdot g)*\} \\ & (sqrt \cdot sqr)* \end{array}$$

Note that the transformation is annotated with the equality used to drive it, written in braces ({ }). Also note that the program above is expressed in point-free form. When BMF programs are expressed in point-free form, transformation rules can be applied as simple re-write rules[10].

BMF has fundamental operations corresponding to the `map`, `reduce` and `scan` constructs provided by Adl. Moreover, all of these functions have efficient sequential and parallel implementations[126].

1.3.1 The role of BMF in this project

A priori, point-free BMF has all of the ingredients needed for targeting to a distributed architecture. It has the explicit data transport of point-free form; it has the requisite sequential and parallel functions and it has a large number of program identities that can be used to transform code. What is now required, for our implementation, is a compiler to translate Adl into point-free BMF, and a compilation process to apply transformations to this BMF code to produce efficient distributed code. Figure 2 gives the structure of the Adl compiler with the stages utilising BMF highlighted in bold. The first two stages of the compiler, the lexer/parser and typechecker are

[7]Where the presence of side-effects and an implicit state that evolves over time make simple, equals-for-equals transformations, only narrowly applicable.

[8]See chapter 3 for a more thorough introduction to BMF.

[9]It should be noted that several BMF variants for functions over aggregate data, such as the map function, exist for a large variety of types including, lists, arrays, bags, sets, trees and others.

[10]Infrequently, these rules have conditions attached which have to be true before the rule can be validly applied.

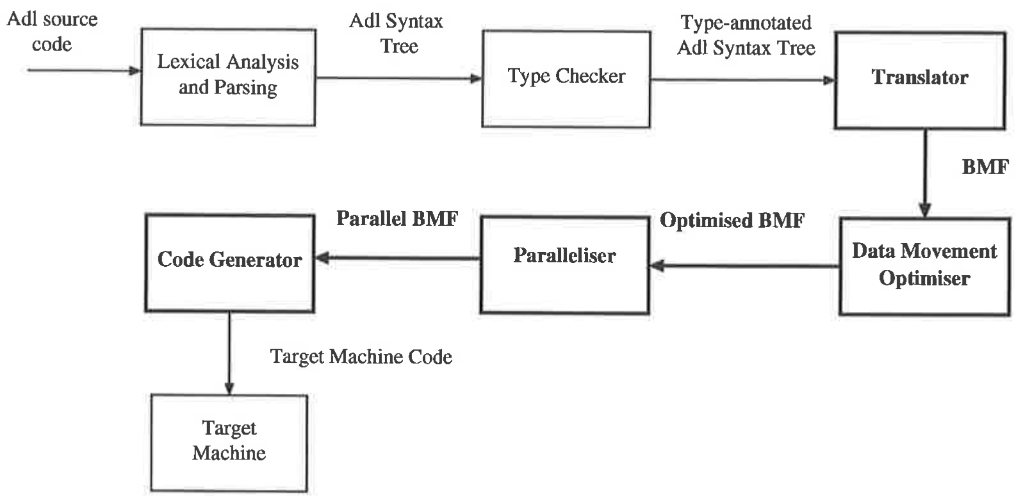


Figure 2. Outline of the Adl implementation. Stages that use or generate point-free BMF code are highlighted in bold.

reasonably conventional. The next four stages are all required to interact with BMF code in some way. The novel aspects of this work lie in these stages. The last stage, the target machine, will have to execute explicitly distributed code. Our very early prototype code generator[111] produces C/MPI code.

Briefly describing the stages that interact with BMF:

- the **translator** takes Adl code and converts it to point-free form with explicit transport of values in the scope to each function[11];

- the **data-movement optimiser** uses incremental transformation of BMF code to drastically reduce the quantity of data transported through the code produced by the translator;

- the **paralleliser** uses incremental transformation to propagate functions with distributed semantics through the code produced by the optimiser;

- the **code generator** translates distributed functions into their equivalents on the target architecture. Sequential functions are compiled down to ordinary imperative code.

Prototype implementations for each of these stages have been produced to demonstrate the feasibility of the pervasive use of BMF as an intermediate form.

[11]This process is made feasible by the functional semantics of Adl. Translation of imperative code into BMF is an open problem.

The novelty of this work is the extent to which the implementation is based on automated transformations in BMF and the techniques developed to assist the process of automation. A brief overview of related work follows.

1.4 Related work

The other part of the Adl project In terms of lineage, the closest relative to this work is a complementary sub-project of the Adl project investigated in [53]. The source language is identical and some tools such as the *partitioning function* (described in chapter 6) are very similar. However, the implementation strategy for this earlier part is based on compilation to a highly efficient, multithreaded abstract machine. Under that implementation, access to remote values is piecemeal and demand-driven. Most static analysis of data requirements is confined to a separate process to determine the partitioning of input data. In contrast, the part of the Adl project described here uses compile-time transformations to consolidate communications and map code to a more low-level platform. Moreover, the intermediate forms used are very different in the two parts of the Adl project.

EL* The Adl implementation described resembles, in some ways, the implementation of EL*[117]. Like Adl, EL* is a small data-parallel language focused on extracting data-parallelism from arrays. EL* permits a restricted form of recursion that can be used to express patterns of computation such as those found in `map` and `reduce`. Access to global variables and array elements from within these definitions is more restricted than the access allowed in Adl. The first stage of the EL* implementation translates EL* source code into FP*, a strongly-typed variant of Backus' point-free FP language[13]. The translation process from Adl to point-free BMF uses similar strategies to the EL* to FP* translator to achieve similar ends. At this stage, however the two implementations diverge. The FP* compiler[144] performs a small amount of transformation in FP* before translating directly to CM-Fortran[134] whose compiler targets the code to a parallel platform. In contrast, the Adl implementation intensively optimises and parallelises the BMF code before translation to an imperative form.

Transformational programming The idea of automating at least some of the program transformation process is a significant driver in the development of the field of transformational programming. Many mechanisable techniques have been developed to optimise parts of sequential and parallel functional code[27, 21, 61, 123, 77, 78]. However, with some notable exceptions[12], the application of these techniques often involves some insight to the problem or at least proof obligations[13]. Partial automation of the transformation process has been carried out using a theorem-prover[98] and, more recently, using an interactive system with a built-in cost model[3]. Mottl[104] surveys a range of techniques for the transformation of functional programs and describes an implementation of a safe automated partial evaluator for a simple strict functional language. This partial evaluator could form an important component of a larger optimisation system.

The Adl implementation is almost fully automated[14]. This level of automation is achieved by applying, with great frequency, a number of simple transformations to point-free programs[15]. To make this process tractable, the Adl compiler makes very heavy use of normalisation[16] to reduce the number of cases that need to be handled. Note that applying transformations in this way will not achieve optimality in most cases, that will typically require some deeper insight into the problem and the application of conditionally beneficial transformations with reference to a cost-model[129, 5] but, even so, efficiency is greatly enhanced.

Data-parallel and Skeleton implementations Adl is, essentially, a single-threaded data-parallel language exploiting familiar data-parallel vector constructs such as `map`, `reduce` and `scan`. This is a similar strategy for exploiting parallelism

[12]These exceptions include, Wadler's de-forestation algorithm[143] and Onoue's HYLO fusion system[108].

[13]Such as proving associativity or distributivity of binary operators.

[14]Though not completely. Adl assumes that the programmers will use associative operators in its `reduce` functions and the Adl compiler does require the programmer to provide some partitioning information for the parallelisation stage.

[15]Adl limits the domain of programs to those most amenable to transformation. In particular, Adl programs are easily converted to a compositional form. Any future introduction of a facility for the expression of unrestricted recursive definitions will inevitably change this situation and perhaps warrant the use of non-trivially applied techniques such as diffusion[78]. It is worth noting that recent work[38] provides a sound basis for expressing recursive definitions in point-free-form but work would still need to be carried out to produce a translation capturing recursive definitions in their most efficient form.

[16]Normalising transformations are often not directly beneficial in terms of efficiency but they make the subsequent application of beneficial transformations easier.

as that used in Data-Parallel C-Extensions, HPF and Connection-Machine Lisp. Unlike DPCE and HPF, Adl has purely functional semantics. Adl has a less explicit communication structure than CM-Lisp.

Adl has more in common with NESL[22, 23] but its implementation strategy differs in that Adl's compiler aims: to minimise data movement at runtime, avoid the use of a substantial run-time system, and explicitly target a distributed machine. This approach comes at the cost of limited support for dynamically evolving nested parallelism which is well-handled by NESL[17].

Crooke[37] implemented core components of BMF using C-Libraries combined with a work-scheduler. Because the implementation language was C, a set of criteria for well-formed programs was also required to avoid non-deterministic behaviour.

In terms of objectives and framework Adl has much in common with various skeleton implementations such as P3L[31], SCL[44], Ektran[71] and the FAN skeleton framework[5][18]. There are significant differences between Adl and these implementations at a more detailed level. Access to variables other than the immediate parameters of the skeleton functions is more implicit in Adl than it is in most skeleton implementations. Adl allows arbitrary access to elements of vectors in global scope, a feature not commonly found in skeleton implementations. Adl offers fewer skeletons than either SCL or P3L and requires more partitioning information from the programmer than P3L which uses profiling information to determine a good partitioning for input data. Like Adl, many skeleton implementations permit the nesting of aggregate functions though the framework for carrying this nesting through to the parallel platform, in some implementations such as P3L and Ektran, is more refined than that used in the Adl implementation. Finally, amenability to transformation at the level of the skeleton functions is a motivator for skeletons work[5, 12, 25], this is also one of the drivers behind the Adl implementation though at this point, Adl makes more pervasive use of program transformations at a lower level[19] and less at a higher-level than other skeleton-based platforms.

[17]The Adl implementation can support nested parallelism but data distribution functions have to be statically determined. A short overview of the application of nested parallelism to Adl is given in Appendix E

[18]To name a few, see chapter 2 of[71] for a more comprehensive overview.

[19]Transformations in the Adl implementation are mainly focused on rationalising the movement of data and, subsequently, on propagating parallelism through code. In the current implementation, there is a limited focus on transforming skeletons to tune parallel performance. Application of such transforms often requires reference to a cost model and some characterisation of input data. Though we have developed a detailed dynamic cost model in the form of a simulator it has not yet been

This concludes our very brief introductory survey of related work. More detailed surveys of relevant work are given at the end of each chapter of this report.

1.5 Preview

This report describes the various stages developed in the implementation the Adl language. Chapter 2 describes the Adl language and provides some examples of its use. Chapter 3 describes the point-free dialect of BMF code used as an intermediate form in the Adl implementation. Chapter 4 gives a formal description of the Adl to BMF translation process and measures the efficiency of code thus produced. Chapter 5 describes the salient features of the optimisation process to increase the efficiency of BMF code through incremental transformation. Chapter 6 defines the process for parallelising code and canvasses issues affecting code-generation. Finally, chapter 7 summarises the findings of our work and presents ideas for future work on this project.

integrated into the transformation system.

Chapter 2

Adl

Adl is the source language for the Adl implementation. This chapter is a brief, informal, introduction to Adl and its primary features. The chapter has three sections. Section 2.1 is an overview of Adl. Section 2.2 very briefly lists possible enhancements in future versions of Adl. Finally, section 2.3 reviews the main features presented in the chapter.

2.1 The Adl language

Adl is a simple experimental language expressive enough to build, without undue difficulty, a variety of applications. Adl is designed to encourage programming in terms of parallelisable aggregate operations.

2.1.1 Background

Adl was first proposed by Roe[119] as a vehicle for experimentation in high-performance functional computing. Two Adl implementations, targeted to distributed architectures, have been developed[7]. The first implementation[53] is multi-threaded, using latency hiding, caching, and various other techniques to minimise communications costs. The second stream, the implementation described here, uses transformational programming to develop parallel implementation with aggregated communications.

In developing both of these implementations Adl was found to be a good medium for the development of small[1] numerically-oriented applications.

2.1.2 Main features

The main features of Adl are:

Functional semantics: Adl is referentially transparent which helps make parallel programming, and implementation, easier.

A strong type system: types are monomorphically instantiated at compile time for more efficient compilation.

Strict/Eager order of evaluation: permitting easier extraction of data-parallelism. No currying is permitted. All functions are monadic (single-valued).

Aggregate types: a vector type for holding an arbitrary number of values of uniform type and a tuple type for holding a fixed number of values of mixed type. Nesting of both tuples and vectors is permitted.

Second-order vector primitives: such as `map`, `reduce` and `scan` for the expression of potential parallelism over vectors.

Limited type polymorphism and overloading: syntactic polymorphism is supported allowing functions to be monomorphically instantiated to the applied types.

Pattern matching: irrefutable patterns are used for easy access to tuple elements.

No recursion: strengthens role for vector operators.

Implicit parallelism: programmer need not be directly concerned with parallelism[2].

[1]and possibly large, though this has not yet been tested.

[2]In Adl most vector functions have the potential to execute in parallel. The programmer is aware, that by using these functions, that parallelism may be exploited. Note that the expression of parallelism is limited to these functions which makes parallelism restricted as well as implicit according to the taxonomy of[130].

```
% A program to add a constant to every element
% of a vector.

my_const := 2;

main a: vof int :=
  let
    add_const x := x + my_const
  in
    map (add_const,a)
  endlet
?
```

Figure 3. A simple Adl program to add a constant to every element of a vector.

Other features include familiar facilities for introducing local scope and declarations and for random access to vector elements. The features above are not set in stone. The restriction on recursion is likely to be lifted in future implementations but, it should be noted that, even with this restriction, Adl is still quite an expressive programming platform.

2.1.3 Program layout

Adl programs consist of a series of function and value declarations ending with a function declaration. Execution of the program begins in the body of this last function declaration. Figure 3 shows a simple Adl program to add two to every element of vector. The execution of this program begins at the `main` function. The formal parameter to `main` is the input value to the program. In this case, the input value is a vector of integers (`vof int`). No other type declarations are required[3] with the type of each function being inferred by propagating the types of input values and constants. The `let` construct introduces a new local scope. In this case the scope contains just one function declaration, `add_const`. The body of the `let` is the starting point of program execution. It contains the primitive function: `map` which takes an input tuple consisting of a function name and a vector value and applies the function to each element of the vector value. Informally:

[3]Though the programmer is free to add type annotations to any other value and/or expression.

```
let
  a := 4;
  b := a + 5;
  c := b - 3;
  b := b + c
in
  a + b
endlet
```

Figure 4. `let` expression with a declaration sequence containing repeated declarations of `b`. This expression evaluates to 21.

$$\texttt{map}\ (f, [x_0, x_1, \ldots, x_{n-1}]) = [f\ x_0, f\ x_1, \ldots, f\ x_{n-1}]$$

As a final note, comments start with an `%` and extend to the end of the line and programs are always terminated with a `?` character.

2.1.4 Declaration sequences

The outermost scope of the program and the first part of any `let` expression consists of a declaration sequence. Declaration sequences can contain one or more function and/or value declarations. Adl enforces a rule that a value is not in scope until immediately after its declaration. It remains visible until the end of the enclosing expression or until another declaration of a value with the same name obscures it. The effect of these rules can be seen in the Adl expression shown in figure 4. The variable environment in scope from the body of the `let` expression, `a + b`, is:

$$[\mathsf{b} \mapsto 15, \mathsf{c} \mapsto 6, \mathsf{b} \mapsto 9, \mathsf{a} \mapsto 4]$$

The variable `b` appears appears twice in this environment but the first instance obscures the second, giving `a + b` = 21. Semantically, Adl declaration sequences are similar to the `let*` construct in Scheme[1][4].

Functions, scope and recursion Adl is statically scoped so functions are evaluated in a closure containing their environment at the time of declaration. Because this environment can contain only values declared prior to the function there can be no recursive or mutually recursive function definitions. To illustrate, consider

[4]Another equivalent form is a nested series of `let` expressions, each binding a single value.

```
odd x := if x = 1 then
           true
         else
           not(even (x-1))      %error ‘‘even’’ not in scope
         endif;
even x := if x = 0 then
           true
         else
           not(odd (x-1))
         endif;
```

Figure 5. An invalid Adl declaration sequence. The reference to **even** in the definition of **odd** is undefined because **even** not in scope at this point.

the, invalid, Adl function declarations in figure 5 to determine if a natural number is even or odd. These declarations will fail to compile because **even** is not in scope from the declaration of **odd**. The same scoping rule also prevents functions from calling themselves.

Higher-order functions In common with languages such as SISAL[55] and NESL[22] Adl provides second order functions such as **map** and **reduce**, but does not allow programmers to define their own higher order functions. This restriction is not an impediment to writing the numerical computations over aggregates that are the focus of this work. However, lifting this restriction may be considered for future implementations.

2.1.5 Types

Adl supports three primitive scalar types:

`int`: integer values.

`real`: floating point values[5].

`bool`: boolean values.

Integer literals are any sequence of numerals, optionally prepended with "-", without a decimal point. Real literals look like integer literals except with a decimal point

[5]assumed to be equivalent to `double` in C.

and at least one numeral to the right of the decimal point. Boolean literals are `true` and `false`. Adl provides a broad range of overloaded arithmetic and comparator functions on `int` and `real` values and some trigonometric functions on `real` values. Two coercion functions, `int()` and `float()` are provided to allow conversions between values of numeric type. Logic primitives such as `and`, `or` and `not` are provided for boolean values.

Adl supports two aggregate types, vectors and tuples. These types, and their functions, are described in turn.

Vectors and their functions Vector literals are comma-separated sequences of zero or more expressions enclosed in square brackets. Examples of valid vector literals include: `[1,-2,3]`, `[true]`, `[1.2,-4.0]`, `[sqr(3.0),-2.3]`, `[(2,true),(0,false)]` and `[]`. Vectors can be nested so, `[[],[3,2,1]]` and `[[true],[false,true,false]]` are valid vectors. All elements of vectors must be of the same type so literals such as: `[1,[2,3]]` and `[3.4,2]` are not allowed.

vector functions There are a number of primitive vector functions including:

`length`: written `#`, returns the number of elements in a vector. Example: `# [1,2,3]` $= 3$.

`index`: an infix function, written `!` to randomly access a vector element. Example `[1,2,3]!0` $= 1$.

`iota`: takes a integer n and produces a list of consecutive integers from 0 to $n-1$. Mechanism for dynamic allocation of vectors. Example: `iota 4` $= [0,1,2,3]$.

`map`: a function taking a pair consisting of a function `f` and a vector `v` and producing a vector containing the results of applying the function to each element of the vector. Example: `map (plus,[(1,1),(2,2),(3,3)])` $= [2,4,6]$ where `plus (x,y) = x + y`.

`reduce`: takes a triple $(\oplus, z, v)$ consisting of a binary function $\oplus$, a value z and a vector v. `reduce` returns z if $v = []$; v_1 if $v = [v_1]$; and $v_1 \oplus \ldots \oplus v_{n-1}$ if $v = [v_1, \ldots, v_{n-1}]$. Example: `reduce(plus,0,[1,2,3])` $= 6$.

```
%concatenate two vectors a and b
concat (a:vof int,b: vof int) :=
   let
     la := #a;
     lb := #b;
     f x :=
       if x < la then
        a!x
      else
        b!(x-la)
      endif
  in
    map (f,iota (la + lb))
  endlet
?
```

Figure 6. An Adl program using both `map` and `iota` to concatenate two input vectors.

`scan`: takes a pair[6]: $(\oplus, v)$, consisting of a binary function $\oplus$ and a vector v. `scan` returns $[\,]$ if $v = [\,]$; v_1 if $v = [v_1]$; and $[v_1, \ldots, v_1 \oplus \ldots \oplus v_{n-1}]$ if $v = [v_1, \ldots, v_{n-1}]$. Example: `scan(plus,[1,2,3])` $= [1, 3, 6]$.

The semantics of `length` and `index` are self-evident. Figure 6 shows a program that uses both `map` and `iota` to concatenate two vectors. This usage of `iota`, as a generator of indices, to guide a subsequent `map`, is quite common.

Vector functions are easily combined. The program in figure 7 uses both `map` and `reduce` to calculate the sum of squares of a vector.

Vector functions may also be nested. The program in figure 8 adds two to every element of a nested input vector.

The `scan` function is useful when intermediate values in a calculation are required. The program in figure 9 calculates the running maximum of a vector of floating-point numbers. It bears mentioning that all vector operations that can have a parallel implementation such as `map`, `reduce`, and `scan` will, potentially, be compiled to parallel code. This means that the programmer must use an associative binary function in `reduce` and `scan`. Failure to do so will lead to non-deterministic results.

[6]Earlier versions of Adl had a version of `scan` that took an additional zero-element in its input arguments. This argument has been eliminated in this version.

```
% calculate the sum of squares
%of an input vector
sum_squares (a:vof integer) :=
  let
     plus (a,b) = a + b;
     sqr  a     = a * a
  in
     reduce(plus,0,map(sqr,a))
  endlet
?
```

Figure 7. An Adl program to calculate the sum of squares of an input vector using `map` and `reduce`.

```
% A program to add a constant to every element
% of a nested input vector.

my_const := 2;

map_map_addconst  a: vof vof int :=
  let
    add_const x := x + my_const;
    f x = map(add_const, x)
  in
    map (f,a)
  endlet
?
```

Figure 8. An Adl program that adds a constant to each element of a nested input vector.

```
% running maximum of a vector of reals

running_max a: vof real :=
  let
    max (x,y) = if x > y then x else y endif;
  in
    scan (max,a)
  endlet
?
```

Figure 9. An Adl program to calculate the running maximum of a vector of floating point numbers.

Where the programmer wishes to use a non-associative operator in `reduce` and `scan` the non-parallel variants: `reducel`,`scanl` (for left-reduce/scan) or `reducer` and `scanr` (right-reduce/scan) must be used[7]. Additionally, a function `reducep` is provided for functions that do not have a convenient zero-element[8]. `reducep` does not take a zero-element but insists upon a non-empty input vector. `reducep` is potentially parallel.

This concludes the description of Adl vectors and vector functions. A brief overview of the Adl tuple type follows.

tuples Tuples are fixed-length aggregates of mixed type. In Adl programs, tuples are specified as literals such as `(2,3)` and `(true,a,5.0)`. Tuples can be nested as in `((a,b),(3,c))` or `((4,5.0,false),b)`. Literals are the sole mechanism for expressing and constructing tuples. An example performing such construction is the function:

```
vec_to_pair a := (a!0,a!1)
```

using a literal to construct a pair out of the first two elements of an input vector.

Members of tuples in Adl are accessed through pattern matching. For example,

$$first(a, b) := a;$$

is a polymorphic function to project the first element from a tuple. Patterns do not need to specify all of the detail in the input tuple. For example, given a function `f`

[7]The Adl compiler does not enforce this use but non-determinism ensues if the programmer uses a non-associative function in `reduce` and/or `scan`.

[8]An example of such an operation is: `left (x,y) := x;`.

with the input type:

```
((int,bool),vof real)
```

any one of the following declarations is valid:

```
f ((a,b),c) := ...;
f (x,y) := ...;
f z := ...;
```

The choice adopted by the programmer is dictated by the parts of the tuple explicitly accessed on the right-hand-side of the function.

All patterns used in Adl must be *irrefutable*. A pattern is irrefutable if it cannot fail to match. The patterns above are all irrefutable. In contrast, the pattern

```
(a,b,a))
```

is refutable because it can fail to match when its first and third elements are not the same. The pattern

```
(a,[c])
```

is refutable because it can fail to match if its second element is not a vector of length one. Adl patterns are used solely for access to tuple elements whereas refutable patterns can also be used for case-analysis. Adl uses `if` expressions rather than patterns for case-analysis.

As a concrete example of the use of tuples and pattern matching, figure 10 shows a program which zips and then unzips a pair of input vectors. The program is an identity function, leaving the input unchanged[9], but it serves to demonstrate the role of pattern matching and tuple literals.

This concludes the explanation of Adl types and their primitive operations. Conditional expressions and iteration are considered next.

2.1.6 Conditional expressions and iteration

Conditional evaluation is expressed using the:

if *predicate* **then** *consequent* **else** *alternative* **endif**

construct which evaluates the boolean expression *predicate* and evaluates *consequent* if it is true or *alternative* if it is false. Examples containing `if` expressions have already been shown.

[9]Assuming that both input vectors are the same length.

```
% a program to zip and then unzip a
% pair of input vectors. Leaving them
% unchanged.
% Precondition: vectors have the same length

zip (a,b) :=
  let
    f x := (a!x, b!x)
  in
    map (f,iota (# a))
  endlet;

unzip a :=
  let
    first (x,y) := x;
    second (x,y) := y
  in
    (map (first,a), map (second, a))
  endlet;

zip_unzip (a:vof int, b:vof int) :=
  unzip(zip(a,b))
?
```

Figure 10. An Adl program to zip and then unzip a pair of input vectors, leaving the input unchanged.

Iteration is expressed using `while` functions. `while` functions take the form:

`while`(*state_transformer*, *predicate_function*, *state*)

This expression applies the *state_transformer* to the *state* to produce a new *state* while the *predicate_function*, applied to *state*, is true. `while` is used whenever the bounds of iteration cannot easily be determined prior to the beginning of iteration. Figure 11 shows an Adl program for calculating an approximation of the square root of a floating-point input value, using Newton's method. The `while` loop will terminate, for positive input values, after a small number of iterations.

This concludes the overview of the Adl language in the current implementation. A brief description of possible enhancements for future versions of Adl follows.

```
% calculate the square root of a positive
% input value

epsilon := 0.00001;

newtons a:real :=
  let
    abs x := if x > 0 then x else -x endif;
    finished x := abs((x * x) - a) < epsilon;
    next_guess x := (x + (a/x))/2.0
  in
    while (finished,next_guess,1.0)
  endlet
?
```

Figure 11. An Adl program using `while` to calculate the square root of a floating point input value using Newton's method.

2.2 Future enhancements

There are a number of enhancements both minor and major that are likely to appear in future versions of Adl. Perhaps the easiest enhancement is the use of syntax-level transformations (*syntactic sugar*) to increase conciseness. Applications of syntactic sugar include:

Operator substitution: allowing the programmer to use expressions like `reduce(+,0,[1,2,3])` instead of the more awkward `reduce(plus,0,[1,2,3])`, which requires the programmer to define `plus`.

Operator sectioning: allowing the programmer to use operator sections like: `(+1)` in contexts such as: `map((+1),[3,4,5])`.

Refutable patterns: allowing the programmer to use patterns for case analysis in function definitions.

Adl would also be enhanced with support for more types. Useful additions include a primitive character type and a type for multi-dimensional vectors with fixed lengths in each dimension[10].

[10]There is much scope to leverage information relating to the shape as well as the basic type of data structures. Jay uses the concept of shape-checking in his language FiSh[86].

The addition of recursion will increase the ease with which many functions are defined. Recursion will need to be supported by primitives allowing incremental construction and destruction of the vector data-type. This will probably entail adding an explicit `concatenate` (++) or cons primitive.

Finally, the addition of a facility for modules would increase scalability and re-usability in an Adl software system. Any module system will need to cater for the propagation of types through the program.

2.3 Summary

Adl is a simple strict functional language with a few primitive base types and aggregate types for vectors and tuples. Adl has constructs supporting: the introduction of local scope, easy access to elements of vectors and tuples, conditional evaluation and iteration. Adl also admits a limited amount of polymorphism to facilitate some code re-use within programs. A number of second-order operations on vectors such as `map`, `reduce` and `scan` are provided to structure programs and serve as potential sources of parallelism. Adl is quite expressive in its current form but there is scope for enhancement with features such as a richer type system and the admission of recursive definitions in future implementations. As it stands, Adl is a good source language for this experimental implementation.

Chapter 3

Bird-Meertens Formalism

Bird-Meertens formalism [18, 58] (BMF) plays a central role in the Adl implementation. This chapter introduces the sub-set of BMF used and its purpose in this implementation. The chapter is divided into three parts. Section 3.1 introduces the salient features of BMF, section 3.2 briefly defines the role of BMF as a medium for program improvement in this work, and, finally, section 3.3 very briefly summarises this chapter.

3.1 Introduction to Bird-Meertens Formalism

In essence, Bird-Meertens Formalism (BMF) is functional programming optimised for the purpose of program transformation[1].

The notation of BMF is loosely defined and extensible. BMF code is not restricted to point-free form but it is at its most transformable when functions are written in point-free form (see [38] for a good exposition on this topic). Functions written in point-free form have no formal parameters and contain no names other than those referring to other functions. There are no variables[2], in the conventional sense, in point-free programs.

For a concrete illustration of point-free form consider the following conventional definition of *abs*, a function to generate the absolute value of its input:

$$abs\ x = sqrt(sqr\ x)$$

[1]Alternative names relating to BMF include *squiggol*, *point-free-form* and *compositional-form*.

[2]In this chapter, the word *variable* means a named place for storing a value generated at run-time.

In point-free form we define it thus:

$$abs = sqrt \cdot sqr$$

where $\cdot$ stands for function composition. In point-free form, we are not permitted to use variables, such as x in the example above, to reference values we have put aside for later use. The only names used in point-free programs are those of functions defined prior to run-time.

Point-free form ensures that producers of data and consumers of data are juxtaposed. This spatial locality paves the way for code to be transformed through simple rewrites using equalities[3]. Together, the open ended notation, and the open-ended set of equalities make up a rich programming calculus, applicable to a number of aggregate types. This calculus is BMF. The remaining parts of this section review the aspects of this calculus relevant to this work[4] starting with a simple introduction to the transformation process.

3.1.1 Some introductory transformations

As an example of the application of equalities, consider the following function definitions:

$$\begin{array}{rcl} abs & = & sqrt \cdot sqr \\ sqr & = & \times \cdot (\mathsf{id}, \mathsf{id})^{\circ} \end{array}$$

a simple transformation, corresponding to no more than an in-lining, or instantiation, of the right-hand-side of the equations is:

$$\begin{array}{ll} & abs \\ = & \quad \{\textit{Definition: abs}\} \\ & sqrt \cdot sqr \\ = & \quad \{\textit{Definition sqr}\} \\ & sqrt \cdot \times \cdot (\mathsf{id}, \mathsf{id})^{\circ} \end{array}$$

Note that all substitutions are equals-for-equals substitutions. Also note that each step is annotated with the equality applied (written in braces). These annotations are a commentary of the transformational proof. Every BMF transformation comes

[3]This is in contrast to the transformation processes for more conventional notations which require some book-keeping to trace data-dependencies through variables.

[4]We use only a subset of the rich variety of constructs available in BMF and point-free programming in general, the reader is referred to [20] and [39].

with its own proof. A core motivation for BMF is this audit-trail of proof as the program is transformed and, hopefully, improved.

The $(\mathsf{id}, \mathsf{id})^\circ$ function in the program above is new. The $(\ ,\ldots,\ )^\circ$, or alltup[5] function, is a second-order function that applies its constituent functions to a copy of its input, forming a tuple consisting of the results. So, for example:

$$(f, g)^\circ\ v$$

will return the pair: $(f(v), g(v))$. The id[6] function is the identity function so: $(\mathsf{id}, \mathsf{id})^\circ v$ returns (v, v).

The transformations above illustrate the methodology of BMF but they have very narrow applicability. The equality[7]

$$sqr = \times \cdot (\mathsf{id}, \mathsf{id})^\circ$$

can only be used to convert the name sqr to its long-hand equivalent or vice-versa.

Much more general transformations exist. Some of these are now reviewed.

3.1.2 General Transformations and Program Structures

Though narrow equalities, such as the ones we have just seen, are an important part of BMF, the most useful and powerful equalities are those that apply to many functions. As an example, given a function π_1 that extracts the first element of an input pair, the following equality:

$$\pi_1 \cdot (f, g)^\circ = f$$

can be applied to effect dead-code elimination. Its applicability is not dependent on the contents of f or g[8], making it very general. A counterpart of the above equality

[5]The name "all-applied-to for tuples" is derivative of the "all-applied-to" construct for lists ($[\,]^\circ$) which appears in Bird's seminal work on BMF[18](section 2.4). There seems to be no agreed-upon notation for alltup. alltup is a generalisation of the *pair* function and the *split* function $<\,,\,>$, both mentioned by Bird in [20]. It is also a generalisation of the $\triangle$ operator described in [39] and similar to a construct used by Gorlatch[63] and Roe[120]. All of these functions map an object in a category into a product.

[6]primitive BMF functions are written in sans-serif font.

[7]The equality states that sqr is a function that duplicates its argument and then multiplies the duplicates.

[8]This must be qualified with note that if there is some valid input value v for which $g(v) = \perp$ and $f(v) \neq \perp$ then the termination properties of the program will change. This is rarely an impediment to transformation in this implementation.

is:

$$\pi_2 \cdot (f, g)^\circ = g$$

Other powerful equalities involving alltup functions are:

$$f \cdot (g, h)^\circ = (f \cdot g, f \cdot h)^\circ$$

and

$$(f, g)^\circ \cdot h = (f \cdot h, g \cdot h)^\circ$$

both of which show how other functions can be absorbed into alltup functions.

Equalities, like the ones above, can be used to prove other, more specific, equalities. For instance, the equality

$$(f \cdot \pi_1, g \cdot \pi_2)^\circ \cdot (h, k)^\circ = (f \cdot h, g \cdot k)^\circ$$

can be proved in the following way:

$$\begin{array}{ll}
 & (f \cdot \pi_1, g \cdot \pi_2)^\circ \cdot (h, k)^\circ \\
= & \qquad \{f \cdot (g, h)^\circ = (f \cdot g, f \cdot h)^\circ\} \\
 & (f \cdot \pi_1 \cdot (h, k)^\circ, g \cdot \pi_2 \cdot (h, k)^\circ)^\circ \\
= & \qquad \{\pi_1 \cdot (f, g)^\circ = f\} \\
 & (f \cdot h, g \cdot \pi_2 \cdot (h, k)^\circ)^\circ \\
= & \qquad \{\pi_2 \cdot (f, g)^\circ = g\} \\
 & (f \cdot h, g \cdot k)^\circ
\end{array}$$

Any equalities generated go into our tool-set for transforming programs. Also note that not all equalities have to be generated using previously defined equalities. Equalities can also be generated from known properties of functions using any proof technique at hand[9].

Because alltup functions can assume an infinite number of arities, producing pairs, triples, quadruples, quintuples and so on, there is an infinite set of equalities that apply to them. Fortunately, it is not difficult to systematically generate and apply such equalities[10].

alltup functions, their corresponding π functions and their attendant equalities are concerned with the creation and use of tuples. A different set of functions and

[9]Unfortunately, such proofs sometimes require insight which inhibits mechanisation of the transformational process.

[10]This implementation uses such systematic techniques during tuple optimisation defined in chapter 5

equalities apply to list-like aggregates. These aggregates, functions and equalities are briefly reviewed next.

3.1.3 Functions working with list-like data

Certain aggregate datatypes, including cons-lists, concatenate-lists, arrays, sets, and bags[11] have the common charateristic of being containers for values of a single type[12]. BMF theories have been developed for each of these, and a number of other, aggregate types. Functions such as map and reduce apply quite naturally to these types and many, very useful, equalities apply to these, and other related, functions. The data-type that underpins the Adl implementation is the concatenate list. Concatenate lists are defined by the constructors:

$$\begin{array}{lcl} [\,]_K & = & [\,] \\ [\cdot]\,x & = & [x] \\ +\!\!+\; xs\, ys & = & xs +\!\!+\, ys \end{array}$$

Where $[]_K$ is a function from *unit* to the list type, $[\cdot]$ (*make_singleton*) makes a singleton list out of an element of the base type, and the $+\!\!+$ function concatenates two lists. A key property of this type is the associativity of $+\!\!+$ which allows the lists to be decomposed and processed in parallel[13]. All of the list functions and equalities that follow are defined on these concatenate-lists. For consistency with the corresponding datatype in Adl, concatenate-lists will henceforth be referred to as *vectors*.

The most important vector functions are map and reduce. map, written $*$, applies the preceding function to each element of a vector. So, for example:

$$sqr*$$

will square each element of an input vector. reduce, written /, inserts the preceding binary function between each element of a vector and evaluates the resulting

[11]Trees and graphs also fall into this category.

[12]Contrast this with tuples, which have a fixed number of elements of mixed type.

[13]Contrast this with cons-lists, which have to be pulled apart one element at a time.

expression. In addition, reduce often takes an additional value, denoted the zero-element, that is returned in the event that the input vector is empty. So, for example:

$$+/_0$$

will find the sum of an input vector and return 0 in the event that the input vector is empty. Another example using reduce is:

$$+\!\!+/_{[\,]}$$

which removes a level of nesting from a nested input vector[14]. The associativity of the application of the binary operation is arbitrary. This means an application such as:

$$+\!\!+/_{[\,]}[[1],[2],[3,4]]$$

could produce:

$$([1]+\!\!+[2])+\!\!+[3,4]$$

or it could produce:

$$[1]+\!\!+([2]+\!\!+[3,4])$$

This is not a problem when the binary operator is associative as above. However directed versions of reduce, $\not\rightarrow$ or $\not\leftarrow$ are required when the binary operator is non-associative[15].

Many general transformations apply to map and reduce. One rule, specialising the more general *promotion theorem* [58], is

$$f*\cdot+\!\!+/_{[\,]} = +\!\!+/_{[\,]}\cdot f**$$

which says, on the LHS, that flattening a nested list and applying f to each element of the less-nested list is the same, on the RHS, as applying f to every element of the nested list and then flattening it. Viewing the above equation as a transformation from the LHS to the RHS is the same as pushing $+\!\!+/_{[\,]}$ leftwards through $f*$.

A second specialisation of the promotion theorem, applying to reduce, is

$$\oplus/_e\cdot+\!\!+/_{[\,]} = \oplus/_e\cdot(\oplus/_e)*$$

which says (LHS) flattening a nested list and then reducing that list with $\oplus$, is the same as (RHS) reducing all of the inner elements of the nested list with $\oplus$ and then

[14]This function is often called *flatten*.

[15]Such operations are inherently sequential. Some optimisations in BMF such as the application of Horner's rule[18] produce such directed reductions, impacting on the potential for parallelism.

reducing the resulting list with $\oplus$. Applying the equality above from left-to-right can be viewed as a transformation that pushes $+\!\!+/_{[]}$ leftwards through $\oplus/_e$, absorbing the former in the process.

We will re-visit the above equalities in chapter 6 when we describe parallelisation of BMF programs.

There are many other identities. One intuitive identity is

$$f * \cdot g* = (f \cdot g)*$$

which describes how two map operations can be merged into one. The application of this equality from left to right is equivalent to merging two loops that apply a function to each element of a vector.

Transformations to mixed list and alltup functions can be applied independently as in the example:

$$\begin{array}{ll} & (f*, g*)^\circ \cdot h* \\ = & \qquad \{(f,g)^\circ \cdot h = (f \cdot h, g \cdot h)^\circ\} \\ & (f * \cdot h*, g * \cdot h*)^\circ \\ = & \qquad \{f * .g* = (f.g) * (\mathit{applied\ twice})\} \\ & ((f.g)*, (g.h)*)^\circ \end{array}$$

The identity:

$$\mathsf{zip}.(f * .\pi_1, g * .\pi_2)^\circ = ((f.\pi_1, g.\pi_2)^\circ) * .\mathsf{zip}$$

is interesting because it shows how the levels of nesting for alltup and map functions can be swapped. zip is a function that takes a pair of lists $([x_0, \ldots, x_{n-1}], [y_0, \ldots, y_{n-1}])$ and converts it into a list of pairs $[(x_0, y_0), \ldots, (x_{n-1}, y_{n-1})]$. The identity above shows how moving a zip function can be used to consolidate two map functions (LHS) into one map function (RHS)[16].

[16]There are generalisations of zip that convert arbitrary length tuples of lists into a list of arbitrary length tuples. One of these, in called $\mathtt{trans}_k$, was used in FP*[145] and implemented as a primitive. Another version was implemented in the source code of a demonstration program for Henk[110], a language with a Pure type system, implemented by Roorda[121]. Such a generalisation would make a useful future addition to the BMF dialect used in the Adl project.

A similar transformation applies to reduce:

$$(\oplus/_e \cdot \pi_1, \otimes/_d \cdot \pi_2)^\circ = (\oplus \cdot (\pi_1 \cdot \pi_1, \pi_1 \cdot \pi_2)^\circ, \otimes \cdot (\pi_2 \cdot \pi_1)^\circ, (\pi_2 \cdot \pi_2)^\circ)^\circ /_{(e,d)^\circ} \cdot \mathsf{zip}$$

where

$$\# \cdot \pi_1 = \# \cdot \pi_2$$

merging two separate reductions (LHS) into a single reduction (RHS)[17]. The equality above is conditional. It only works if we can guarantee the length of its first argument is the same as the length of the second argument. This fact is denoted by the predicate after the *where* clause. # is the list-length operator in BMF. Several of the transformations used in this work are based on such conditional equalities.

This concludes our brief discussion on some basic list transformations. There are many other transformations on many other functions. The ones above serve as examples, but they are important examples. The proof systems underlying these equalities are to be found in category theory. A strong exposition of the categorical foundations of BMF can be found in [20] and some more recent work expanding the applicability of point-free BMF can be found in [40].

3.1.4 Homomorphisms

Homomorphisms[18] are functions whose structure matches the structure of the type on which they operate. A homomorphism on a list type is a function that has a case for each constructor in the list type. Recall, the constructors for the list type we use in this work are:

$$\begin{array}{lcl} [\,]_K & = & [\,] \\ [\cdot]\, x & = & [x] \\ +\!\!+\; xs\, ys & = & xs +\!\!+ ys \end{array}$$

A homomorphism on concatenate-lists is a function h with the following structure:

$$\begin{array}{lcl} h\,[\,] & = & e \\ h\,[x] & = & f x \\ h(xs +\!\!+ ys) & = & (h\, xs) \oplus (h\, ys) \end{array}$$

[17]The explanation for the function $(\pi_1 \cdot \pi_1, \pi_1 \cdot \pi_2)^\circ$ to the right of $\oplus$ is that reduce will present each instance of its binary operator with a pair of operands. In the RHS of the equality above the pair presented to $\oplus \cdot (\pi_1 \cdot \pi_1, \pi_1 \cdot \pi_2)^\circ$ has the form: $((p_1, q_1), (p_2, q_2))$ where p_1 and p_2 are the values destined for the binary $\oplus$ function. $(\pi_1 \cdot \pi_1, \pi_1 \cdot \pi_2)^\circ$ is the code to extract these two operands to form the pair (p_1, p_2). Similar reasoning applies to the code to the right of the $\otimes$ function.

[18]Homomorphisms on an initial algebra are called *catamorphisms* [58, 20, 40]. The homomorphisms discussed here are also catamorphisms.

where each line matches a constructor of the concatenate list. The value e is a constant that is produced if the input list to h is empty. f is applied to each element of the list. $\oplus$ is used to join list elements after they have been processed by f. e is assumed to be the left and right identity of $\oplus$, that is: $e \oplus x = x \oplus e = x$.

The notation above is a clumsy way of writing homomorphisms. A simpler way is to simply list the values of e, f and $\oplus$:

$$h = (\!| e, f, \oplus |\!)$$

Homomorphisms can also be uniquely expressed as a combination of map and reduce[19]:

$$h = \oplus/_e \cdot f*$$

3.1.4.1 The importance of homomorphisms

Homomorphisms are important for several reasons including:

Versatility: Homomorphisms can be used to express a wide variety of functions. They are pertinent to many programming problems.

Transformability: map and reduce have many useful equalities that apply to them. The many functions that can be expressed as a homomorphism can be modified/improved by these equalities.

Parallelisability: both map and reduce have obvious and efficient parallel implementations.

Table 1 gives some examples of list homomorphisms. A much bigger class of functions can be expressed as a homomorphism followed by a projection (π function). The existence of these additional functions functions, called near-homomorphisms[36], adds greatly to utility of map and reduce.

Relevance of homomorphisms to the Adl project The Adl language contains no single construct corresponding to a homomorphism and there are no parts of the Adl compiler designed specifically to exploit homomorphisms, yet homomorphisms,

[19]There is a subtle difference between the e in a homomorphism, which is a constant and the e in reduce which is a constant function. In this work, the distinction between constant and constant function is of little importance. The implementation can be easily adjusted to cope with one or the other.

Homomorphism	Description
$sum = \llparenthesis 0, \mathsf{id}, + \rrparenthesis$	The function to find the sum of a list of numbers.
$prod = \llparenthesis 1, \mathsf{id}, \times \rrparenthesis$	The function to find the product of a list of numbers.
$alltrue\ p = \llparenthesis \mathsf{true}, p, \wedge \rrparenthesis$	The function to test if px is true for all elements x of the input list.
$onetrue\ p = \llparenthesis \mathsf{false}, p, \vee \rrparenthesis$	The function to test if px is true for at least one element x of the input list.
$max = \llparenthesis -\infty, \mathsf{id}, \uparrow \rrparenthesis$	The function to find the maximum value in a list of numbers using the $\uparrow$ function to return the maximum of two numbers.
$min = \llparenthesis \infty, \mathsf{id}, \downarrow \rrparenthesis$	The function to find the minimum value in a list of numbers.
$length = \llparenthesis 0, 1, + \rrparenthesis$	The function to find the length of a list.
$rev = \llparenthesis [\,]_K, [\cdot], \widetilde{+\!\!+} \rrparenthesis$	The function to reverse a list using the $\widetilde{+\!\!+}$ (concatenate in reverse) function.
$sort = \llparenthesis [\,]_K, [\cdot], \wedge\!\!\wedge \rrparenthesis$	The function to sort a list using the $\wedge\!\!\wedge$ (merge) function.
$\mathsf{map}\ f = \llparenthesis [\,]_K, ([\cdot] \cdot f), +\!\!+ \rrparenthesis$	Formulation of $\mathsf{map}\ f$ as a homomorphism.
$\mathsf{reduce} \oplus e = \llparenthesis e, \mathsf{id}, \oplus \rrparenthesis$	Formulation of $\mathsf{reduce} \oplus e$ as a homomorphism.
$\mathsf{scan} \oplus e = \llparenthesis [e], [\cdot], \odot \rrparenthesis$	Formulation of $\mathsf{scan} \oplus e$ as a homomorphism where $xs \odot ys = xs +\!\!+ (((last\,xs)\oplus) * ys)$ and $last\,xs$ returns the last element of the vector xs.
$\mathsf{id} = \llparenthesis [\,]_K, [\cdot], +\!\!+ \rrparenthesis$	The identity function on lists.

Table 1. Some list-homomorphisms.

and their near-relatives, are very important to the Adl project. This importance stems from the fact that the Adl language is designed to encourage the programmer to express solutions in terms of aggregate functions including `map` and `reduce`. Any code expressed in terms of either of these two functions inherits their transformability and their potential for parallelism. Moreover, their combination forms a homomorphism which can be used to express the solution to a large variety of problems. The relevance of homomorphisms to Adl stems from the fact that the primitives of Adl can be used to express homomorphisms, along with the fact that homomorphisms can be used to express readily tranformable solutions to many problems.

3.1.4.2 The role of scan

Adl provides a small family of scan primitives but the BMF equivalent: scan[20], written $/\!\!/$, has not yet been mentioned. Informally the semantics of $/\!\!/$ is:

$$\oplus /\!\!/ \; [x_0, x_1, \ldots, x_{n-1}] = [x_0, x_0 \oplus x_1, \ldots, x_0 \oplus x_1 \oplus \ldots \oplus x_{n-1}]$$

scan is a very versatile operation, useful when the intermediate results of a reduction are needed. As indicated in table 1, scan can be implemented in terms of map and reduce but, due to the inefficiency of doing this, it usually kept as a primitive in its own right. There are several useful equalities that apply to scan, which do not bear direct mention here, but a sample of these can be found in [64]. Like reduce, some varieties of scan have a fast parallel implementation. The Adl implementation exploits this parallelism.

This completes the overview of BMF. This description barely scratches the surface. There are many more BMF types, primitives and equalities available. The reader is referred to [20] for a good introduction to the theoretical basis of BMF and the methodology and to [78] for an overview of some more recent developments in optimisation of parallel BMF.

We have also omitted descriptions of several of the BMF functions used in the Adl implementation. These will be introduced in later chapters as they become relevant. A description of the role of BMF in the Adl implementation follows.

3.2 The role of BMF in the Adl project

Figure 12 highlights the role of BMF in the Adl implementation. The places where BMF is produced or consumed are highlighted in black. The primary role of BMF in this project is as a medium for program improvement. The translator emits BMF code which is then improved by the optimiser resulting in faster code. The syntax of optimised code is slightly richer due to new primitives injected to increase the efficiency of code[21]. The parallelisation process injects and propagates implicitly

[20]Sometimes called prefix.

[21]All other things being equal, a richer syntax in an intermediate form is a bad thing: it gives us more cases to handle. Some rationalisation of this language may be possible in future implementations.

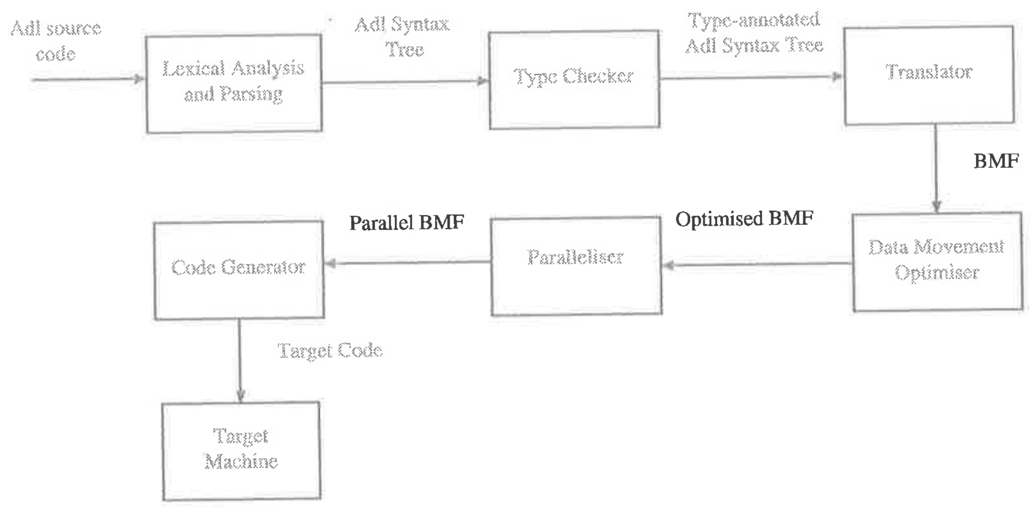


Figure 12. The role of BMF in the Adl project. The places where BMF acts as a bridge between two stages are highlighted in black.

distributed BMF primitives through code[22]. At all stages, all code is expressed completely in point-free form down to the level of BMF primitives. To a large extent, this is made feasible by code-inlining which is made feasible by the absence of recursion. The aim of using BMF in this project is to attempt to exploit, to the maximum possible extent, the transformability of point-free BMF. That is, to find out what techniques are most productive in the automated improvement of such code.

3.3 Summary

This ends our brief introduction to BMF and aspects of the point-free variant used in the Adl implementation. To summarise: BMF is unmatched as a medium for program transformation and improvement; transformations in BMF rely on mathematical equalities, making them safe, and to an extent, mechanisable; it is this safety and mechanisability that we aim to exploit in the Adl implementation. As a final note, it should be said that point-free BMF with its need for explicit transport of values is not an easy notation for programming. This lack of ease motivates the use of a more

[22]In theory, such parallelisation may seem unnecessary since it is possible to directly map the data-flow-graph formed by BMF code and its input onto a machine with an unbounded number of virtual processors. However, this approach rules out any algorithms extract efficiency by recognising the boundary between on-node and distributed processing. As an example, the efficient algorithm for parallel `scan` given in chapter 6 relies upon such knowledge.

friendly source language, Adl, and the implementation of an Adl to BMF translation process. This translation process is the subject of the next chapter.

Chapter 4

Adl to BMF translation

The last two chapters explored two different notations. The first notation, Adl, is a small strict functional language with built-in second-order functions. The second notation, point-free BMF, is a calculus for programming, based on a notation, consisting entirely of functions, glued together by second order functions.

This chapter describes a translation process from Adl to BMF. The first section of this chapter consists of a brief statement of the role of the translator in this project. The second section contains a brief, informal summary of what the translator does. The third, and largest, section defines the translator formally. The detailed explanations in this section can be skipped on a first reading. The fourth section shows examples of code produced by the translator and assesses the efficiency of this code. Finally we briefly summarise related work.

4.1 What the translator does

In this work, the translator has the task of converting between Adl programs and BMF programs. The nature of Adl and BMF, and their role in this work, is summarised below.

Adl and its role Adl and the point-free BMF[1] used in this project share a strict functional semantics and a reliance on second order functions.

[1]Henceforth, we will use term "BMF" to refer to the dialect of point-free BMF used in this project.

The main point of divergence is that Adl, like most languages, supports variables as means of storing run-time data while BMF does not[2]. Variables provide programmers with an abstraction of a, single-assignment, random-access store. Such a store lets the programmer decouple consumers of data from producers of data. With variables, the programmer concentrates more on the actual production and consumption of values and less on how to get values between producers and consumers, the implicit store handles the problem. This separation of concerns makes programming simpler and is largely taken for granted, especially in non-distributed environment where the random access store is efficiently realised in hardware.

We believe it is valuable for programmers to maintain the abstraction of a random-access store even in a distributed environment[3]. Adl, the source language for this implementation does maintain this abstraction.

BMF and its role BMF does not support variables as a means of storing run-time data. In BMF a function producing data and a function consuming that data must be composed with each other. Transport of data is carried out by passing values through sequences of composed functions. From the point of view of the BMF programmer storage exists only fleetingly in the form of the immediate input and output values of the currently executing function.

BMF is easily transformed through the localised application of program identities. Moreover, its primitives have naturally parallel implementations. These two properties make it, a priori, a good candidate for the intermediate form of our implementation.

The translator's task The translator assumes that the Adl source file has been parsed and type-checked before it begins its operations[4]. The translator takes an Adl abstract syntax tree and converts all of it to BMF. The translator is highlighted in black in the map of our implementation shown in figure 13. One goal of this project is

[2]In other words, Adl supports point-wise functional programming while the BMF notation we use is point-free.

[3]There is a strong opposing argument that, for the sake of efficiency, the distinction between access to local values and to remote values should be visible to, and under the control of, the programmer. We believe the need for this distinction is diminished for Adl where it is most convenient for the programmer to use constructs with efficient parallel implementations and the cost of data movement is statically reduced by the compiler.

[4]Parsing and type-checking of Adl programs are reasonably straightforward activities and of little research interest so we omit details of these operations from this document.

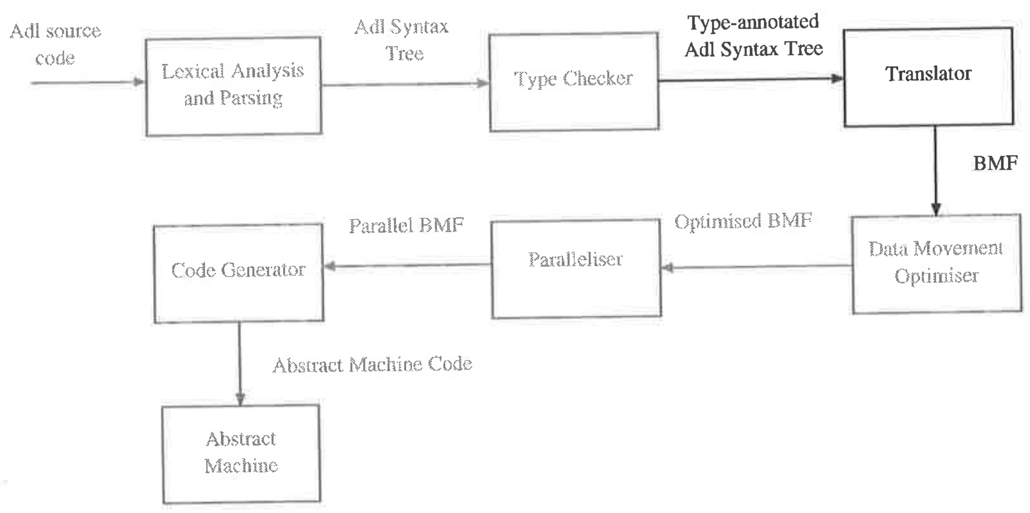


Figure 13. The Adl project with the translation stage highlighted in black.

to leverage the transformational properties of BMF to the maximum possible extent. In keeping with this goal, the translation process was kept as straightforward as possible. Any optimisation of the Adl source or optimisation during the translation process is eschewed in favour of performing that optimisation once the program is expressed in BMF[5].

The primary change wrought by the translation process is the elimination of variables. Other things including the names and semantics of primitive functions are preserved to the maximum possible extent. An overview of the translation process is given next.

4.2 Overview of translation

The Adl to BMF translator's primary task is to change the method of data access from variables to functions. During translation, each reference to a variable is replaced by a function that projects, the value from the values in scope. So, given the program:

```
simple(a:int,b:int) := a
```

our translator produces the BMF code

[5]An open question is what optimisations are more easily performed while the program is is still expressed in Adl. Certainly some simple substitutions and optimisations such as common sub-expression elimination appear to be easily done in Adl. The question of what is best done in Adl and what is best done in BMF is interesting future work.

$$\pi_1$$

that is, the function that returns the first value out of the input tuple.

A slightly more complex example

```
simple2(a:int,b:int) := b - a
```

is translated to the BMF code

$$- \cdot (\pi_2, \pi_1)^{\circ}$$

where the two input arguments are permuted before the subtraction operation is performed.

Declaration sequences In the programs above, the only values in scope are formal parameters. Scope is more complex in Adl programs that contain declaration sequences. Each variable declaration in a sequence introduces a new value to the previous scope. The translator must produce BMF code to transport values in scope to places they are used. The Adl program:

```
not_as_simple(b:int) :=
   let
      a = 20
   in
      b - a
   endlet
```

translates to:

$$- \cdot (\pi_1, \pi_2)^{\circ} \cdot \mathsf{id} \cdot (\mathsf{id}, 20)^{\circ}$$

Reading from right to left, the first part of this program, $(\mathsf{id}, 20)^{\circ}$, wraps the input value with the value 20 to form a tuple

$$(\mathit{val(b)}, 20)$$

where *val(b)* denotes the value that replaces the parameter `b` at runtime. The middle id in the BMF code is an artifact of the translation process. The last part, $- \cdot (\pi_1, \pi_2)^{\circ}$, extracts both values from the input tuple and subtracts one from the other.

Functions in declaration sequences Function declarations are treated differently from variable declarations. Variable declarations are converted immediately into code, whereas no code is produced when a function declaration is encountered. Instead a *closure* is created consisting of the function's code, formal parameters and referencing environment at the time of function declaration. Every time the function is called code for the function is built, using the information in the closure and information about the current referencing environment, and in-lined into the target program[6].

As a concrete example of this process, consider the translation of the code:

```
function_tester a: int :=
  let
    f(x,y) := x + y;
    b := 20
  in
    f(a,b)
  endlet
```

into the BMF code:

$$+ \cdot (\pi_1 \cdot \pi_2, \pi_2 \cdot \pi_2)^\circ \cdot (\mathsf{id}, (\pi_1, \pi_2)^\circ)^\circ \cdot$$
$$\mathsf{id} \cdot (\mathsf{id}, 20)^\circ$$

The start of the program, on the second line above, corresponds to the declaration `b := 20`. There is no code corresponding to the function declaration `f(x,y) := x + y`. Instead the first line:

$$+ \cdot (\pi_1 \cdot \pi_2, \pi_2 \cdot \pi_2)^\circ(\mathsf{id}, (\pi_1, \pi_2)^\circ)^\circ$$

corresponds to the call `f(a,b)`. The first part of this line to be executed:

$$(\mathsf{id}, (\pi_1, \pi_2)^\circ)^\circ$$

appends the values of the actual parameters of `f` to the end of the input tuple. The end of the program, at the beginning of the first line:

$$+ \cdot (\pi_1 \cdot \pi_2, \pi_2 \cdot \pi_2)^\circ$$

is the body of the function f applied to the actual parameters.

The flows of data through the above program are illustrated in figure 14. The

[6]In-lining of calls to recursive functions is problematic. The Adl implementation currently avoids this problem by banning recursive definitions. Translation of recursive functions into BMF is an active research topic[78, 40].

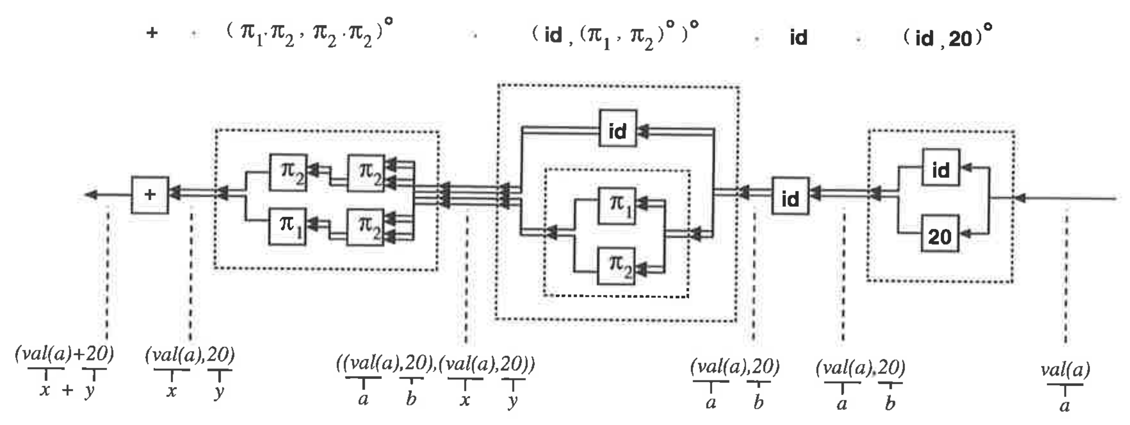

+ · $(\pi_1 \cdot \pi_2, \pi_2 \cdot \pi_2)^\circ$ · $(\mathsf{id}, (\pi_1, \pi_2)^\circ)^\circ$ · id · $(\mathsf{id}, 20)^\circ$
(val(a)+20) x + y
(val(a),20) x y
((val(a),20),(val(a),20)) a b x y
(val(a),20) a b
(val(a),20) a b
val(a) a

Figure 14. The translated code for `function_tester.Adl` (top), a diagrammatic rendering of this code (middle) and a description of the data at various points of execution (bottom).

translated code for `function_tester.Adl` is shown at the top of the diagram. A graphical representation of the translated code is in the middle of the diagram. The text below the diagram shows the set of input/output values at each stage of the program. It is worth noting that, even though the function `f` accesses only the values attached to its formal parameters, the code also transports other copies of these values just in case they might be accessed via the variables `a` and/or `b` inside the body of `f`. This leads us to an important point: the translator has a policy of producing code that transports *every* value that was previously visible to the code of a function to each call of that function. This policy is conservative: the code never fails to transport a value that is needed. The drawback of the policy is that values that are *not* needed are transported along with the ones that are.

The results of the translator's conservative policy are visible in figure 14; the code produces correct results but is more complex than required. In this work, the translator's sole responsibility is producing a *correct* translation. Later on, the optimisation stage of the implementation, described in chapter 5 attempts to minimise the amount of data transported.

The role of types in translation The Adl typechecker monomorphically instantiates and annotates every node in the Adl syntax tree. The translation process does not directly utilise this type information but it is useful for parallelisation and code generation so it is assumed the translator preserves all type information.

This concludes our brief overview of the translation process. Next, we describe the translation process in much more detail, presenting the semantic rules used to build our prototype translator.

4.3 The rules of translation

The prototype translator Adl to BMF is formally defined using Natural Semantics[89]. Natural Semantics is a variety of structural operational semantics (see [74] for a very clear introduction). By using natural semantics we can describe an aspect of a language or a process in terms of *relations* between expressions. These relations are written as rules. A set of rules can be used to prove if a relation holds between expressions. Moreover, if the expressions given to a set of rules is partially undefined, the rules can be used to instantiate the undefined parts.

The rules in this chapter define the relationship between Adl code and the BMF code produced by the translator. The relationship can be written as:

$$A \Rightarrow B$$

where "$\Rightarrow$" means "translates to". If B is unspecified the translator rules instantiate it to the first valid translation of A (assuming one exists). If B is specified then the relationship acts as a predicate that is proven when B is a valid translation of A.

The translator's semantic rules have been implemented and tested using the Centaur[24] system from INRIA. Centaur provides the means to execute a definition in Natural Semantics against expressions in the input syntax of the definitions. In the translator implementation, our rules are applied to Adl programs and BMF code is automatically instantiated on the other side of the rule.

4.3.1 The source and target syntax

The abstract syntax definitions of the source and target languages follow starting with a definition of the Adl abstract syntax.

4.3.1.1 Adl syntax

In Centaur, syntax definitions are divided into two parts. The first part defines the classes of objects, called *Phyla* that appear in the abstract syntax. The Phyla mappings serve to convey the role of each part of the syntax to the reader.

The second part is the abstract syntax itself. It is written in a variant of extended Backus-Naur form. Here, Adl construct names are in `typewriter` font. Phyla names are in capitalised italics. Operators of fixed arity (e.g. `map(`*ID,EXP*`)`) have round brackets. Operators with a list of descendants have square brackets (e.g. `decls[`*DEC**`]`)

A superscript of * inside square brackets indicates zero or more occurrences of the objects of the preceding phyla name. A superscript of $^+$ denotes one or more occurrences[7].

Empty parentheses, following an operator (e.g. `int()`) show that it has no descendants. Other terminal objects whose range of possible values is too large to enumerate (e.g. *REAL*) are defined by "implemented as" clauses. The domain from which these values are drawn is described in English.

The following definition is the complete abstract syntax of Adl so it is quite lengthy[8]. It is included as a reference for the Adl expressions in the translator rules. The most relevant aspect of the syntax for the reader is the allocation of syntax objects to phyla. Phyla names are used to denote the type of expressions in the translator rules.

Phyla

$$\begin{array}{l}
PROG \mapsto \textit{Programs},\ DECLS \mapsto \textit{Declarations},\ DEC \mapsto \textit{Declaration},\\
EXP \mapsto \textit{Expressions},\ PAT \mapsto \textit{Patterns},\ TYPE \mapsto \textit{Type specifier},\\
ID \mapsto \textit{Identifier},\ IDT \mapsto \textit{Optionally typed identifier},\\
BOP \mapsto \textit{Binary operator},\ UN\_OP \mapsto \textit{Unary operator},\ NUM \mapsto \textit{Integer},\\
REAL \mapsto \textit{Real}.
\end{array}$$

[7]Ideally, we would have a $^{+2}$ superscript for items in a tuple but this is not supported by Metal, the Centaur language used to define this syntax.

[8]One reason for this length is the explicit overloading of primitive arithmetic operations. Later versions of this implementation will insert coercion operations during the type-checking phase.

Abstract syntax

```
PROG    ::= pr(DECLS)
DECLS   ::= decls[DEC*]
DEC     ::= vardec(IDT,EXP) | fundec(IDT,PAT,EXP)
EXP     ::= let(DECLS,EXP) | funapp(ID,EXP) | if(EXP,EXP,EXP) |
            while(ID,ID,EXP) | map(ID,EXP) | binop(BOP,EXP,EXP) |
            iota(EXP) | reduce(ID,EXP,EXP) | reducer(ID,EXP,EXP) |
            reducel(ID,EXP,EXP) | reducep(ID,EXP) | reducelp(ID,EXP) |
            reducerp(ID,EXP) | scan(ID,EXP) | scanl(ID,EXP) |
            scanr(ID,EXP) | uop(UN_OP,EXP) | vecl[EXP*] | ID | NUM |
            REAL | tuple[EXP+]
PAT     ::= ID | typed_id(ID,TYPE) | pat_list[PAT*]
TYPE    ::= int() | real() | bool() | vof(TYPE) | type_tuple[TYPE+]
IDT     ::= ID | typed_id(ID,TYPE)
BOP     ::= and() | or() | eq() | eqii() | eqir() | eqri() | eqrr() | eqbb() |
            neq() | neqii() | neqir() | neqir() | neqri() | neqrr() | neqbb() |
            leq() | leqii() | leqir() | leqri() | leqrr() | leqbb() |
            geq() | geqii() | geqir() | geqri() | geqrr() | geqbb() |
            gt() | gtii() | gtir() | gtri() | gtrr() | gtbb() |
            lt() | ltii() | ltir() | ltri() | ltrr() | ltbb() |
            plus() | plusii() | plusir() | plusri() | plusrr() |
            minus() | minusii() | minusir() | minusri() | minusrr() |
            times() | timesii() | timesir() | timesri() | timesrr() |
            div() | divii() | divir() | divri() | divrr() |
            power() | powerii() | powerir() | powerri() | powerrr() |
            index() | intdiv() | mod()
UN_OP   ::= uminus() | uminusi() | uminusr() |
            length() | neg() | round() | float() | trunc() |
            sin() | cos() | tan() | asin() | acos() | atan()
BOOL    ::= true() | false()
ID      ::= id implemented as a string
NUM     ::= num implemented as an integer
REAL    ::= real implemented as a real number
```

4.3.1.2 BMF syntax

The conventions followed in BMF syntax are very similar to those followed for Adl. Primitive BMF functions are defined in sans-serif font to help further distinguish target and source code in the rules[9].

The phyla and abstract syntax definition are each divided into two parts. The first parts are the core of the target language. The second parts are definitions of auxiliary constructs. The auxiliary constructs are used to set up an environment in which the translation can take place, acting as scaffolding for the translation process. Objects defined in the Auxiliary syntax do not appear in the target code produced by the translator.

Finally, the abstract syntax of types defines valid type annotations for functions in BMF programs. All functions in BMF programs are annotated with a type from the *B_FTYPE* phyla. The syntax for types reflects the restriction of user-defined functions to being first-order and uncurried. It should be noted that type annotations are, currently, not exploited by the translator or optimiser phases of this implementation so we leave them implicit in the rules shown in this chapter and the next. Types are exploited during parallelisation so we will see them again in chapter 6.

Phyla

$B\_EXP \mapsto$ *Expressions*, $B\_K \mapsto$ *Constant functions*,
$B\_OP \mapsto$ *Operators*, $B\_INT \mapsto$ *Integers*, $B\_REAL \mapsto$ *Real numbers*

Auxiliary Phyla

$B\_ENV \mapsto$ *Environment*, $B\_VAR \mapsto$ *Variable environment*
$B\_VB \mapsto$ *Variable binding*, $B\_VOD \mapsto$ *Variable or Dummy entry*,
$B\_FUN \mapsto$ *Function environment*, $B\_FB \mapsto$ *Function binding*,
$B\_CLOS \mapsto$ *Function closure*, $B\_NUM \mapsto$ *Number*, $B\_V \mapsto$ *Variable name*

Type Phyla

$B\_FTYPE \mapsto$ *Function types*, $B\_TYPE \mapsto$ *Non-Functional types*,
$B\_TVAR \mapsto$ *Type variables*

[9]Our Centaur implementation does not maintain an explicit distinction between source and target syntax, they are both components of the same syntax definition.

Abstract syntax

B_EXP	::=	b_comp(*B_EXP*,*B_EXP*) | b_alltup[*B_EXP*$^+$] | b_con(*B_K*) |
		b_allvec[*B_EXP**] | b_op(*B_OP*) | b_id() | b_map(*B_EXP*)|
		b_reduce(*B_EXP*,*B_EXP*) | b_reducel(*B_EXP*,*B_EXP*) |
		b_reducer(*B_EXP*,*B_EXP*) | b_reducep(*B_EXP*) | b_reducelp(*B_EXP*) |
		b_reducerp(*B_EXP*) | b_scan(*B_EXP*) | b_scanl(*B_EXP*) |
		b_scanr(*B_EXP*) | b_if(*B_EXP*,*B_EXP*,*B_EXP*) |
		b_while(*B_EXP*,*B_EXP*) | b_addr(*B_NUM*,*B_NUM*) |
B_K	::=	b_true() | b_false() | *B_INT* | *B_REAL*
B_OP	::=	b_iota()| b_and()| b_op()| b_distl()|
		b_eq() | b_eqii() | b_eqir() | b_eqri() | b_eqrr() | b_eqbb() |
		b_neq() | b_neqii() | b_neqir() | b_neqri() | b_neqrr() | b_neqbb() |
		b_le() | b_leii() | b_leir() | b_leri() | b_lerr() | b_lebb() |
		b_ge() | b_geii() | b_geir() | b_geri() | b_gerr() | b_gebb() |
		b_lt() | b_ltii() | b_ltir() | b_ltri() | b_ltrr() | b_ltbb() |
		b_gt() | b_gtii() | b_gtir() | b_gtri() | b_gtrr() | b_gtbb() |
		b_plus() | b_plusii() | b_plusir() | b_plusri() | b_plusrr() |
		b_minus() | b_minusii() | b_minusir() | b_minusri() | b_minusrr() |
		b_times() | b_timesii() | b_timesir() | b_timesri() | b_timesrr() |
		b_div() | b_divii() | b_divir() | b_divri() | b_divrr() |
		b_power() | b_powerii() | b_powerir() | b_powerri() | b_powerrr() |
		b_intdiv() | b_index() | b_mod()
		b_uminius() | b_uminusi() | b_uminusr() |
		b_and() | b_or() | b_length() | b_neg() | b_round() | b_trunc() |
		b_float() | b_sin() | b_cos() | b_tan() | b_asin() | b_acos() | b_atan()
B_NUM	::=	b_num implemented as an *integer*
B_INT	::=	b_int implemented as a *lifted integer*
B_REAL	::=	b_real implemented as a *lifted real number*

Auxiliary abstract syntax

B_ENV	::=	b_env(*B_VAR*,*B_FUN*,*B_NUM*)
B_VAR	::=	b_var[*B_VB**]
B_VB	::=	b_vb(*B_VOD*,*B_EXP*)

B_VOD	::= b_dummy() | *B_V*
B_FUN	::= b_fun[*B_FB**]
B_FB	::= b_fb(*B_V*,*B_CLOS*)
B_CLOS	::= b_clos(*B_EXP*,*B_NUM*,*B_VAR*,*B_VAR*,*B_FUN*)
B_V	::= b_v implemented as a *string*

Abstract syntax of BMF types

B_FTYPE	::= *B_TYPE* → *B_TYPE*
B_TYPE	::= b_integer_t | b_real_t | b_boolean_t | *B_TVAR* | [*B_TYPE*] | (*B_TYPE*,. . . , *B_TYPE*)
B_TVAR	::= b_tv implemented as a *string*

4.3.1.3 Explanatory notes for BMF syntax

BMF Syntax for Integer Constants The syntax contains two types of integer constant. The first type, *B_INT* is the phyla of lifted integers[10]. All objects in this phyla are functions that return an integer value when given an argument of any type. They (and real number constants and boolean constants) are written in sans-serif font.

The second type, *B_NUM* is the phyla of ordinary integers. *B_NUM*'s are used in the auxiliary and also in the target syntax. where *B_NUM*'s are used as hard-wired parameters of the b_addr(,) function.

BMF Syntax for addressing tuple elements b_addr(,) is the way that π functions are written in the abstract syntax. As shown in chapter 3, π functions are the means by which elements of tuples are accessed. To date, we have only examined tuples which are pairs of values:

$$(x, y)$$

where x and y are values of arbitrary type. To access x and y we use the functions π_1 and π_2 respectively. π_1 is written b_addr$(2, 1)$, in the abstract syntax, and π_2 is written b_addr$(2, 2)$. When the arity of the input tuple is greater than 2 the π

[10]A lifted integer is a function that returns the same integer constant irrespective of its input value.

functions accessing that tuple are left-superscripted with the arity of the tuple. So, for example, to access the first, second and third elements of the tuple:

$$(x, y, z)$$

we use the functions ${}^3\pi_1$, ${}^3\pi_2$, and ${}^3\pi_3$ respectively. The corresponding functions in the abstract syntax are $\mathsf{b\_addr}(3, 1)$, $\mathsf{b\_addr}(3, 2)$, and $\mathsf{b\_addr}(3, 3)$ respectively.

As a final but important note, the two values in the brackets of $\mathsf{b\_addr}(\ ,\ )$ functions are fixed at compile-time. In effect, b_addr is a family of functions where individual members are named, at compile time, by placing numbers in the brackets.

BMF Syntax for Function Composition Function composition is expressed using the binary $\mathsf{b\_comp}(\ ,\ )$ operator. Under this regime, sequences of composed functions are built using several b_comp operators. For example the program:

$$f \cdot g \cdot h$$

can be written

$$\mathsf{b\_comp}(f, \mathsf{b\_comp}(g, h))$$

or, it can be written as the different but equivalent BMF expression:

$$\mathsf{b\_comp}(\mathsf{b\_comp}(f, g), h)$$

The flexibility of being able to associate b_comp expressions in any convenient order makes the creation of code by the translator easy. However, there is a cost, incurred at later stages of compilation, in having to cater for multiple versions of an expression. We could avoid multiple versions by defining composition sequences to be lists. This means that $f \cdot g \cdot h$ could be written in only one way:

$$\mathsf{b\_comp}[f, g, h]$$

The compositions as lists approach used in [19](Chapter 12) and [71]. The Adl implementation uses a binary composition instead of compositions as lists because binary composition offers a mechanism for partitioning code without altering the semantics of that code. Such partitionings are heavily exploited by the optimisation process described in the next chapter.

Auxilliary syntax Finally, the auxilliary syntax defines constructs used to construct the environment, containing variable and function bindings, in which the translator rules work. None of the elements of the auxilliary syntax appear in target code. The environment is used by several rules and will be described as the rules are introduced in the remainder of this section.

4.3.2 Interpreting the rules

The following paragraphs contain notes on how to interpret the semantic rules shown in this section. They are also a useful guide to some conventions used throughout the rest of this document.

4.3.2.1 How rules are structured

The environment As stated previously, all of the rules of this chapter take the form of a relationship between Adl expressions and BMF expressions. For the most part, the $\Rightarrow$ symbol is used to denote this relationship. The rule

$$A \Rightarrow B$$

is an assertion that the Adl expression A translates to the BMF expression B.

In most cases A will contain code that references variables and functions. Where variable references exist in A, it cannot be translated without a context to resolve these references. The context we use is an environment constructed from the variable and function declarations currently in scope. In natural semantics, a rule that applies with respect to some environment ENV is written:

$$ENV \vdash A \Rightarrow B$$

This is read "A translates to B in the context of the environment ENV".

Rules with multiple lines The single-line format above is used in the axiomatic rules for translating some terminal expressions in Adl. For most Adl expressions, a translation rule is contingent on other properties. For example, the translation of a variable reference cannot be proved until we can prove that the variable can be found in the environment. More generally, the translation of a non-terminal expression cannot be proved until the translations of its sub-expressions are proved.

For example, to find a valid translation for the expression $exp_1 + exp_2$ we need to find a valid translation for exp_1 and exp_2. These extra conditions are integrated into the rules in the following manner:

$$\frac{ENV \vdash exp_1 \Rightarrow b_1 \qquad ENV \vdash exp_2 \Rightarrow b_2}{ENV \vdash exp_1 + exp_2 \Rightarrow + \cdot (b_1, b_2)^\circ}$$

This reads: "$exp_1 + exp_2$ translates into the BMF expression $+ \cdot (b_1, b_2)^\circ$ provided there is a translation from exp_1 to some BMF expression b_1 and there is a translation from exp_2 to some BMF expression b_2, all in the context of the environment ENV". When reading a rule, we read the last line first followed by the other lines, the *premises*, in descending order. In this document, we refer to the last line of a rule as the *conclusion* of that rule.

4.3.2.2 Naming and style conventions for the translator rules

Where possible, naming conventions follow those used in the syntax of each language.

Variable names All variables in the rules are written in lower-case italics. This applies whether they denote Adl code or BMF functions. Where possible, the variable name conforms to the phylum forming its domain e.g. a variable holding BMF expressions will usually be written as some variation of *b_exp*. Where the name of the phylum cannot be used the phylum of a variable can be determined from the syntactic context.

The names of different variables of the same phylum within a rule are distinguished using subscripts and/or prime (′) symbols.

Rule Sets Centaur requires rules to be grouped into *Rule-sets*. A rule-set is a collection of rules with a single purpose. The translator definition contains several rule-sets and two of these, *Trans* and *Translate*, are described in this chapter.

Judgements Translator rules within each set, depending on their role, have different formats. In Centaur, formats for rules are declared at the top of each rule set. The conclusion of each rule in a set must adhere to one of the formats declared for that set. These formats are called *judgements*. The *Translate* rule-set has a judgement of the form:

$$B\_ENV \vdash EXP \Rightarrow B\_EXP$$

which declares "this rule-set contains some rules whose conclusion starts with a BMF environment followed by a $\vdash$ symbol, followed by an Adl expression followed by a $\Rightarrow$ symbol, and finishes with a BMF expression". *Translate* also contains another judgement for rules for handling declarations:

$$B\_ENV \vdash DECLS : B\_ENV, B\_EXP$$

Different judgements are distinguished by different symbols and different phyla names.

References to other judgements References, in the premises of a rule, to a judgement different from the one found in the conclusion of that rule are described as *calls*. Where the call is to a judgement from a different rule-set, the call is qualified by the name of the other rule-set, written in italics. Calls to different judgements in the same rule-set do not need to be qualified.

Brackets Constructs, in both Adl and BMF, are written according to their structure. Where operators have fixed arity, arguments are encapsulated in round brackets. List-like structures use square brackets. List-like structures can be composed and decomposed in the same way as cons-lists. That is, lists consist of a head and a tail and we can recursively access the tail of a list until we encounter the empty list. Commas can be used as shorthand notation in some instances. The following are equivalent.

```
vector[2,3]
vector[2.vector[3]]
vector[2.vector[3.vector[]]]
```

Note that, for the sake of our implementation, tuples are treated as list-like structures by the semantic rules.

Constructs of zero-arity are usually followed by a pair of matching round brackets. Integer and real constants are exceptions to these rules. They are written as a name followed by a variable or literal (no parentheses). For example: `num` 3, `b_num` N.

4.3.2.3 Miscellaneous terminology

The following describes a variety of phrases of relevance to the description of the translator and to explanations of concepts throughout the rest of this document.

The notion of run-time In this chapter and in the remainder of this document there are references made to the behaviour of target code at "run-time". For example: "At run-time this (code) will be given input representing the current environment...". This notion is potentially ambiguous because the nature of the target code can change completely through the stages of compilation through to the target machine.

In this chapter, run-time is when the BMF code, taken directly from the translator, is applied to its input values.

Upstream-downstream BMF code is read from right-to-left and from bottom-to-top. This unconventional reading order renders terms like "beginning", "end", "start", and "finish" slightly unclear. To avoid ambiguity, in the remainder of this document we refer to relative locations in BMF code using the terms "upstream" and "downstream".

Code that is closer to the point at which the BMF program begins execution is *upstream*. Code closer to the point at which execution ends is called *downstream*. As an example of this usage, consider the program:

$$\oplus/_e \cdot h * \cdot g * \cdot f * \cdot \mathsf{zip} \cdot (\pi_2, \pi_1)^\circ$$

The function $f*$ is upstream of $g*$ but downstream of zip. The most-upstream function is $(\pi_2, \pi_1)^\circ$ where both the π_2 and π_1 functions are equally upstream. Finally, the most-downstream part of the program is $\oplus/_e$.

The meaning of *body* The explanations throughout the rest of this document use the term *body* to refer to the core component of a BMF function or of an Adl expression. The ways we use this term are listed below.

`let` body: In an expression `let` < *decls* > `in` < *body* > `endlet` the part labelled < *body* > is the body of the `let` expression.

function body: In Adl, the function body is the code following the := in a function declaration. In BMF, it is the BMF code produced from the Adl code making up the body of a function declaration. The BMF code to add a function's actual parameters to that function's input tuple is *not* part of that function's body.

map body: Used in the context of BMF to denote the code embedded inside a map expression. The map body corresponds to the part labelled $< body >$ in map($< body >$) (BMF abstract syntax) or in ($< body >$)$*$ (BMF concrete syntax).

Where the term body is used outside of the contexts above, it should be taken to mean the *core* function/expression of the construct in question.

Vectors vs. Concatenate-Lists Throughout the rest of this report we use the term vector to refer to concatenate lists.

4.3.2.4 Layout of rules in this chapter

The rules are defined in figures. Each figure contains rules for related constructs. A figure, depending on space constraints, may or may not contain all of the rules in a rule-set. For the purposes of documentation, we also display the structure of each judgement used in a figure in the top-right-hand corner of the figure (see the top-right-hand corner of figure 15 on page 57 for an example).

The translator rules follow.

4.3.3 The rules

The translator's definition is composed of approximately 20 different rule-sets. Most of these sets are small, auxiliary, rule-sets containing only a handful of rules and one or two judgements. In this chapter we describe two rule sets in detail:

Trans: a small rule-set to pre-process Adl programs into a single expression plus an environment.

Translate: the core of the translator, turning an Adl expression into a BMF expression in the context of a given environment.

Rules in these two rule-sets call judgements from auxiliary rule-sets. The role of each auxiliary rule set is briefly described as it is encountered but details of these rules are not provided. A more complete description of the translation process can be found in [6]. We start with a description of *Trans*, the pre-processing judgement.

Set *Trans* **is**

$$\boxed{\begin{array}{l} \vdash PROG = B\_EXP \\ \vdash DECLS \rightarrow DECLS, DEC \end{array}}$$

Top rule

$$\frac{\begin{array}{c} \vdash decls \rightarrow decls', \texttt{fun\_dec}(v,pat,exp) \\ Rename\_parameters(\vdash pat \rightarrow pat') \\ Rename\_bound\_vars(pat, exp \rightarrow exp') \\ Extract\_bvar(pat' \rightarrow b\_var) \qquad Length\_var(b\_var \rightarrow b\_num) \\ Translate(\mathsf{b\_env}(b\_var,\mathsf{b\_fun}[],b\_num) \vdash \texttt{let}(decls', exp') \Rightarrow b\_exp) \end{array}}{\vdash \texttt{pr}(decls) = b\_exp} \qquad (1)$$

Splitting declarations

$$\vdash \texttt{decls[fun\_dec(}v,pat,exp\texttt{)]} \rightarrow \texttt{decls[]}, \texttt{fun\_dec(}v,pat,exp\texttt{)} \qquad (2)$$

$$\frac{\vdash \texttt{decls[}dec'\texttt{]}.decls \rightarrow decls', dec''}{\vdash \texttt{decls[}dec.\texttt{decls[}dec'.decls\texttt{]]} \rightarrow \texttt{decls[}dec.decls'\texttt{]}, dec''} \qquad (3)$$

End *Trans*

Figure 15. Top level rules. Programs are formed from a list of declarations

4.3.3.1 The *Trans* rule set

The rules of *Trans* are shown in figure 15. Their purpose is convert the Adl source program from a sequence of declarations to a single let-expression and then pass this expression, along with an environment containing bindings for formal parameters of the function that is the program's entry point, onto the *Translate* rule-set for conversion to BMF.

Trans takes, as input, a program of the form:

```
⟨defn 1⟩
⟨defn 2⟩
...
⟨defn n-1⟩
⟨id⟩ ⟨pat⟩ := ⟨exp⟩
```

where the < `defn 1` > to < `defn n-1` > are arbitrary function and variable declarations and the last declaration:

$$< \texttt{id} >< \texttt{pat} > \texttt{:=} < \texttt{exp} >$$

is the function declaration that is the starting-point of program execution. *Trans* converts the input above to a single `let` expression of the form:

```
let
        < defn 1 >
        < defn 2 >
        ...
        < defn n-1 >
in
        < exp >
endlet
```

plus an environment constructed from the formal parameter list < `pat` >. The first three premises of rule 1 handle the re-formatting of code described above. The fourth and fifth premises convert a modified parameter list into a variable environment under which the `let` expression above can be translated. Both the `let` expression and the environment are given to the *Translate* rule set in the last premise to obtain a BMF translation. We now look at the premises to rule 1 in more detail.

The first premise is a call to the second judgement of *Trans* (defined by rules 2 and 3). These rules split the declaration sequence into two parts. The first part

```
< defn 1 > < defn 2 > ... < defn n-1 >
```

(labelled *decls'* in rule 1) goes directly into the `let` expression given to *Translate*. The second part

```
< id > < pat > := < exp >
```

is processed further. First, the formal parameters < `pat` >, are renamed by a call to the auxiliary rule-set *Rename_parameters*[11]. The formal parameters are renamed to prevent a program like:

```
a := 2;
b := a + 5;
f(a,b) := a - b
```

being converted to:

[11]Note again, that details of auxiliary rule-sets, such as *Rename_parameters*, will not be given here.

```
let
         a := 2;
         b := a + 5
in
         a - b
endlet
```

where a and b, in the `let` body, now refer to incorrect values. A correct translation:

```
let
         a := 2;
         b := a + 5
in
         _a - _b
endlet
```

is achieved by using *Rename_bound_vars* to prefix underscores to the names of variables bound to the formal parameters. We also prefix underscores to the names of parameters in < `pat` > to produce < `pat'` > using *Rename_parameters*.

Now, we have the program modified to a single let expression and a renamed formal parameter list < `pat'` >. The fourth and fifth premises of rule 1 build components of an environment from < `pat'` >. *Extract_bvar* constructs a variable environment from the parameter pattern.

The variable environment The variable environment is a set of bindings from variable names to BMF code. The variable environment extracted from the parameter list `(_a,_b)`, by*Extract_bvar*, is:

$$\mathsf{b\_var}[\mathsf{b\_vb}(\_a, \mathsf{id} \cdot \pi_1), \mathsf{b\_vb}(\_b, \mathsf{id} \cdot \pi_2)]$$

Extract_bvar is able to convert arbitrarily nested parameter lists[12] to a *flat* variable environment. The fifth premise, *Length_var* finds the length of the variable environment. This length value plays an important role in adjusting variable references during the translation of function calls[13] (see rule 15 on page 68).

[12] Recall from chapter 2 that patterns in Adl can be nested to any depth.

[13] The length value is a measure of the level of nesting of the input tuple to the current function. This value is used during the reconstruction of a variable environment for a function closure when that function is called.

Set *Translate* **is**

$$\boxed{\begin{array}{l} B\_ENV \vdash EXP \Rightarrow B\_EXP \\ B\_ENV \vdash DECLS : B\_ENV, B\_EXP \end{array}}$$

Constants

$$b\_env \vdash \texttt{int}\ i \Rightarrow \mathsf{b\_con}(\mathsf{b\_int}\ i) \tag{4}$$

$$b\_env \vdash \texttt{real}\ r \Rightarrow \mathsf{b\_con}(\mathsf{b\_real}\ r) \tag{5}$$

$$b\_env \vdash \texttt{boolean(true())} \Rightarrow \mathsf{b\_con}(\mathsf{b\_true}()) \tag{6}$$

$$b\_env \vdash \texttt{boolean(false())} \Rightarrow \mathsf{b\_con}(\mathsf{b\_false}()) \tag{7}$$

Figure 16. Translator rules for constants

The last premise injects the variable environment and its length into a fully-fledged environment and then feeds this environment and the modified code into a call to *Translate* in order to complete the task of translation of the new `let` expression.

4.3.3.2 The *Translate* rule-set

The *Translate* rule-set is large, so we have broken it up up into several figures. Each figure contains a set of related rules. Each element of Adl syntax has a corresponding rule for BMF translation. The rules of *Trans* correspond to these elements of syntax. We start with the simplest rules first and progress to more complex ones.

Constants The rules for Adl integer, real and boolean literals appear in figure 16. Each of these rules is an axiom. Their validity can be proved without resort to premises or even to the environment. In all of these rules the Adl literal is injected into a BMF construct, b_con denoting a constant function. That is, the Adl literal is lifted to a BMF function.

Binary operators Figure 17 contains the rule for translating applications of binary operators. The translation process follows the structure of the binary expression. The two arguments to the operator are translated, in turn, by the rule's first two premises. The third premise converts the Adl operator to its BMF equivalent by the *Convert_op* rule-set. As might be surmised, *Convert_op* defines a trivial mapping between Adl operators and BMF operators. The last step of translation, in the right-hand-side of the rules conclusion, combines the results of the three premises.

$\boxed{B\_ENV \vdash EXP \Rightarrow B\_EXP}$

Binary operators

$$\frac{\begin{array}{c} b\_env \vdash exp_1 \Rightarrow b\_exp_1 \\ b\_env \vdash exp_2 \Rightarrow b\_exp_2 \\ Convert\_op(bop \rightarrow b\_op) \end{array}}{\begin{array}{l} b\_env \vdash \mathtt{binop}(bop, exp_1, exp_2) \Rightarrow \\ \qquad \mathsf{b\_comp}(\mathsf{b\_op}(b\_op), \mathsf{b\_alltup}[b\_exp_1, b\_exp_2]) \end{array}} \quad (8)$$

Figure 17. Translator rule for binary operators

$\boxed{B\_ENV \vdash EXP \Rightarrow B\_EXP}$

Vector literals

$$b\_env \vdash \mathtt{vecl}\,[] \Rightarrow \mathsf{b\_allvec}[] \quad (9)$$

$$\frac{b\_env \vdash exp \Rightarrow b\_exp \qquad b\_env \vdash vecl \Rightarrow b\_allvec}{b\_env \vdash \mathtt{vecl}\,[exp.vecl] \Rightarrow \mathsf{b\_allvec}[b\_exp.b\_allvec]} \quad (10)$$

Figure 18. Translator rules for vector literals

Unary operators and the Iota function Unary operators and the iota function are translated in a very similar manner to binary operators. The only difference being one less premise to handle the single operand.

Vector literals Figure 18 contains the rules responsible for translating vector literals. In Centaur, list-like structures, like vectors and tuples are recursively defined. The two translation rules mirror this recursive structure.

Rule 9 handles empty vectors. It generates an empty b_allvec expression which is, essentially, a lifted constant which returns an empty vector given any input.

Rule 10 traverses a non-empty vector. The first premise translates the first item. The second is a recursive call to handle the remainder of the vector. The conclusion combines the results two premises into a single b_allvec function.

Tuples Tuples are handled in a very similar ways to vector literals. The only difference is that tuples cannot be of zero-length so we have a rule to handle singleton

$\boxed{B\_ENV \vdash EXP \Rightarrow B\_EXP}$

If expressions

$$\frac{\begin{array}{c} b\_env \vdash exp_1 \Rightarrow b\_exp_1 \\ b\_env \vdash exp_2 \Rightarrow b\_exp_2 \\ b\_env \vdash exp_3 \Rightarrow b\_exp_3 \end{array}}{b\_env \vdash \texttt{if}(exp_1,exp_2,exp_3) \Rightarrow \mathsf{if}(b\_exp_1,b\_exp_2,b\_exp_3)} \quad (11)$$

Figure 19. Translator rule for conditional expressions

tuples rather than zero-length tuples[14].

Iota functions The `iota` function is translated in the same manner as unary functions.

If expressions Figure 19 contains the rule for translating Adl `if` expressions to BMF if expressions. This is our first encounter with BMF if expressions in this report. These expressions have the semantics:

$$\begin{aligned} &\mathsf{if}(b\_exp_1, b\_exp_2, b\_exp_3)\ v \\ &\qquad = b\_exp_2\ v, \boldsymbol{if}\,(b\_exp_1\ v = true) \\ &\qquad = b\_exp_3\ v, \boldsymbol{otherwise} \end{aligned}$$

Rule 11 has three premises that translate the sub-expressions, exp_1, exp_2, and exp_3 into their corresponding BMF functions.

Let expressions `let` expressions introduce variables and functions into scope. In the following pages we describe the translation of let expressions, declaration sequences, variable references and function applications. These descriptions outline the core task of the translator: the replacement of the role of variables with functions.

`let` expressions consist of two parts. The first part, the declaration sequence, brings new values and functions into scope. The second part is the body of the `let` expression. The body is an expression that has access to the new environment created

[14]The rule for singleton tuples is not mutually exclusive of the rule for longer tuples. This overlap has dire consequences for efficiency when the translator fails to find a solution first time and goes on a determined search for an alternate proof (backtracking). In short, when something goes wrong, and rules are not mutually exclusive, Centaur can take a very long time to give up.

$$\boxed{\begin{array}{l} B\_ENV \vdash EXP \Rightarrow B\_EXP \\ B\_ENV \vdash DECLS : B\_ENV, B\_EXP \end{array}}$$

Let Expressions

$$\frac{b\_env \vdash exp \Rightarrow b\_exp}{b\_env \vdash \texttt{let(decls[]}, exp) \Rightarrow b\_exp} \quad (12)$$

$$\frac{\begin{array}{c} b\_env \vdash decls : b\_env', b\_exp \\ b\_env' \vdash e \Rightarrow b\_exp' \end{array}}{b\_env \vdash \texttt{let(}decls,e\texttt{)} \Rightarrow \mathsf{b\_comp}(b\_exp', b\_exp)} \quad (13)$$

Figure 20. Translator rule for let expressions

by the declaration sequence. The result of evaluating the body is the result of the entire `let` expression.

The top-level rules for handling `let` expressions are shown in figure 20. Rule 12 is pure book-keeping; it handles `let` expressions with an empty declaration sequence. These empty sequences arise only as a result of pre-processing programs consisting of a single function declaration. All other `let` expressions, those with non-empty declaration sequences, are handled by rule 13. Its first premise processes the declaration sequence, producing both an updated environment and a BMF expression. The second premise evaluates the body of the `let` expression in the context of the new environment. The BMF code, produced by each premise, is combined in the rule's conclusion.

Declaration sequences Declaration sequences bring new values and functions into scope. The process translating a declaration sequence must satisfy three obligations. These are:

1. to produce BMF code for each *variable* declaration encountered.

2. to produce BMF code to ensure the transport of values generated by variable declarations to later parts of the program.

3. to keep track of new values as they enter into scope and their relative position in the environment.

In fulfilling these obligations, the translation process turns declaration sequences into new BMF code, with facilities to transport values, and a modified environment.

The rules for handling declaration sequences include detailed definitions for keeping track of offsets within the environment. This level of detail belies the simplicity of the process that these rules define so, in the interest of clarity, we describe the translation of declaration sequences informally, using examples.

Example: variable declarations The translation of the declaration sequence:

```
a := 4;
b := a+3
```

in the environment:

$$\mathsf{b\_env}(\mathit{oldvar}, \mathit{oldfun}, \mathit{oldvarlength})$$

where *oldvar*, *oldfun* and *oldvarlength* are the pre-existing variable environment, function environment and variable environment length, respectively, produces the code

$$(\mathsf{id}, + \cdot (\pi_2, 3)^\circ)^\circ \cdot (\mathsf{id}, 4)^\circ$$

and the new environment:

$$\mathsf{b\_env}(\mathsf{b\_var}[\mathsf{b\_vb}(\mathsf{a}, \mathsf{id}), \mathsf{b\_vb}(\mathsf{b}, \mathsf{id}), \mathit{oldvar}], \mathit{oldfun}, \mathit{oldvarlength} + 2)$$

The new code wraps its input value in tuples containing the constants 4 and 3. That is:

$$\begin{array}{ll} (\mathsf{id}, + \cdot (\pi_2, 3)^\circ)^\circ \cdot (\mathsf{id}, 4)^\circ\ \mathit{val} & = \\ (\mathsf{id}, + \cdot (\pi_2, 3)^\circ)^\circ\ (\mathit{val}, 4) & = \\ ((\mathit{val}, 4), 7) & \end{array}$$

The new environment is the old environment with two entries added to the variable environment and the length of the environment incremented by 2. The id expressions bound to a and b, in the new variable environment, are composed with the BMF expression produced when these values are referenced (see the description of variable translation, starting on page 66, for a summary of how references are handled).

How the environment is updated The environment is updated incrementally as the declaration sequence is processed. In the last example

```
a := 4;
b := a+3
```

the declaration of the variable a produced the updated environment

$$\mathsf{b\_env}(\mathsf{b\_var}[\mathsf{b\_vb}(\mathsf{a},\mathsf{id}), \mathit{oldvar}], \mathit{oldfun}, \mathit{oldvarlength}+1)$$

in which to evaluate the RHS of b:

```
a+3
```

Note that the declaration of b is not visible from the RHS of a's declaration. That is, the translator does not permit forward references. Only variables and functions that have already been declared can be referenced; declarations further down the sequence are invisible. This policy preserves the semantics of Adl declaration sequences.

Example: function declarations Function declarations generate new entries in the function environment, they do not, directly, generate code. The translation of the declaration sequence:

$$f(x,y) := x+y;$$

in the environment:

$$\mathsf{b\_env}(\mathit{oldvar}, \mathit{oldfun}, \mathit{oldvarlength})$$

creates the new environment:

$$\mathsf{b\_env}(\mathit{oldvar}, \mathsf{b\_fun}[\mathsf{b\_fb}(\mathtt{f}, \mathit{fclos}), \mathit{oldfun}], \mathit{oldvarlength})$$

The new entry in the function environment

$$\mathsf{b\_fb}(\mathtt{f}, \mathit{fclos})$$

binds the variable f to *fclos* where *fclos* is a *closure* for the function f. A closure contains all of the information needed to translate a future call to a function. A closure contains a function's code, its formal parameter list, and the contents of the environment at the time of declaration. In our example, *fclos* takes the form

$$\mathsf{b\_clos}(\texttt{x + y}, \mathit{oldvarlength}, \mathit{oldvar}, \mathsf{b\_var}[\mathsf{b\_vb}(\texttt{x}, \pi_1), \mathsf{b\_vb}(\texttt{y}, \pi_2)], \mathit{oldfun})$$

where *oldvar*, *oldfun*, and *oldvarlength* are taken from the old environment and the function's code

```
x + y
```

and formal parameter list

$$\mathsf{b\_var}[\mathsf{b\_vb}(\texttt{x}, \pi_1), \mathsf{b\_vb}(\texttt{y}, \pi_2)]$$

make up the rest of the closure. The π_1 and π_2 functions bound to `x` and `y` indicate that the actual parameters to `f` will be stored in a tuple and that the π_1 and π_2 functions, respectively, will be part of the code used to access these parameters. We describe how closures are used in the section on function application, starting on page 67.

Variables Variable references are resolved by translating Adl variables to the BMF code that is required to access the value which is bound to the variable. It is assumed that those translation rules applied to earlier declarations have produced code to transport all values in scope to this current part of the program inside a nested tuple whose structure is accurately described by the current variable environment. The translator rule for variables need only exploit the information in this environment to build the correct sequence of addressing functions to access the desired value from the nested input tuple.

A simple example We illustrate how access code is built using the following program:

```
f(x :int, y:  int) :=
   let
      z := x + y
   in
      y - z
   endlet
```

$$\boxed{B\_ENV \vdash EXP \Rightarrow B\_EXP}$$

Variable names

$$\frac{Lookup\_var(b\_var \vdash \mathtt{id}x \rightarrow b\_exp)}{\mathsf{b\_env}(b\_var, b\_fun, b\_num) \vdash \mathtt{id}x \Rightarrow b\_exp} \quad (14)$$

Figure 21. Translator rule for variables

Assume the translation process has reached the variable reference `y`, in the expression `y-z`, in the body of the `let` expression. At the time of `y`'s translation, the variable environment looks like:

$$\mathsf{b\_var}[\mathsf{b\_vb}(\mathtt{z}, \mathsf{id}), \mathsf{b\_vb}(\mathtt{x}, \pi_1), \mathsf{b\_vb}(\mathtt{y}, \pi_2)]$$

This environment is an encoding of the values transported to this part of the program

$$(\,(val(\mathtt{x}), val(\mathtt{y})),\; val(\mathtt{z}))$$

where the term $val(var)$ stands for the value attached to the variable var.

To construct code for variable reference, the translator counts its way[15] through the variable environment, composing π functions to its result until a binding with the variable's name is reached. The code produced for the reference to `y` is:

$$\pi_2 \cdot \pi_1$$

which can be verified as correct by applying it to the input tuple:

$$\begin{array}{lll} \pi_2 \cdot \pi_1 & ((val(\mathtt{x}), val(\mathtt{y})), val(\mathtt{z})) & \Rightarrow \\ \pi_2 & (val(\mathtt{x}), val(\mathtt{y})) & \Rightarrow \\ val(\mathtt{y}) & & \end{array}$$

The foregoing gives an intuition of the construction process for access code. The rule to handle this construction is shown in figure 21. Most of the construction task, as outlined above, is handled by the call to the rule-set *Lookup_var* in the premise.

Function applications The translation of a function application

$$\mathtt{f} < \textit{parameter-list} >$$

requires five steps:

[15]There is some detail hidden in the term "counts its way" due to the occassional presence of tuple values in the input tuple.

$\boxed{B\_ENV \vdash EXP \Rightarrow B\_EXP}$

Function application

$$\frac{\begin{array}{c}\mathsf{b\_env}(b\_var, b\_fun, b\_num) \vdash exp \Rightarrow b\_exp \\ Lookup\_funenv(b\_fun \vdash id \rightarrow \mathsf{b\_clos}(exp', b\_num', b\_var', b\_var'', b\_fun')) \\ Build\_new\_env(b\_num, b\_num', b\_var', b\_var'', b\_fun' : b\_env') \\ b\_env' \vdash exp' \Rightarrow b\_exp'\end{array}}{\begin{array}{l}\mathsf{b\_env}(b\_var, b\_fun, b\_num) \vdash \mathtt{fun\_app}(id, exp) \Rightarrow \\ \qquad \mathsf{b\_comp}(b\_exp', \mathsf{b\_alltup}[\mathsf{b\_id}(), b\_exp])\end{array}} \tag{15}$$

Figure 22. Translator rule for function applications

1. Translate the < *parameter-list* > into BMF code.
2. Search for the closure of `f` in the function environment.
3. Build a new environment in which to translate the code of the function.
4. Translate the function in the context of the new environment.
5. Compose the translated function code with the code produced from the translation of < *parameter-list* >.

The rule for function applications is shown in figure 22. The first four steps above correspond to the four premises of rule 15. The last step is carried out by the right-hand-side of the rule's conclusion. We illustrate these steps by way of example.

Example: translating a program containing function application In this example we translate the program:

```
f a:int :=
  let
     g b := b + a;
     c   := 4;
     d   := c + 3
  in
     g d
  endlet
```

The expression yielding the result of this program is the function application

```
g d
```

in the body of the `let` expression. Pre-processing of the program creates the environment

$$\mathsf{b\_env}(\mathsf{b\_var}[\mathsf{b\_vb}(\mathsf{a}, \mathsf{id})], \mathsf{b\_fun}[], 1)$$

in which to translate the expression:

```
let
   g b := b + a;
   c   := 4;
   d   := c + 3
in
   g d
endlet
```

The translation of the declaration sequence

```
g b := b + a;
c   := 4;
d   := c + 3
```

creates the code

$$(\mathsf{id}, + \cdot (\mathsf{id} \cdot \pi_2, 3)^\circ)^\circ \cdot (\mathsf{id}, 4)^\circ$$

and updates the environment to:

$$\mathsf{b\_env}(\mathsf{b\_var}[\mathsf{b\_vb}(\mathsf{d}, \mathsf{id}), \mathsf{b\_vb}(\mathsf{c}, \mathsf{id}), \mathsf{b\_vb}(\mathsf{a}, \mathsf{id})], \mathsf{b\_fun}[\mathsf{b\_fb}(\mathsf{g}, gclos)], 3)$$

where *gclos* is (see the section on function declarations on page 65 for a description of closures):

$$\begin{aligned}\mathsf{b\_clos}(\ & \mathtt{b\ +\ a}, 1, \\ & \mathsf{b\_var}[\mathsf{b\_vb}(\mathsf{a}, \mathsf{id})], \\ & \mathsf{b\_var}[\mathsf{b\_vb}(\mathsf{b}, \mathsf{id})], \\ & \mathsf{b\_fun}[\,])\end{aligned}$$

Now, the translator is ready to process the function application:

$$\mathsf{g\ d}$$

The first step is to translate the actual parameter list d into BMF code which, in this case is:

$$\pi_2$$

This function will, in the rule's conclusion, be embedded in an alltup expression producing:

$$(\mathsf{id}, \pi_2)^\circ$$

The second step of translation is finding the closure for f in the function environment. This step is trivial. The third step is to build the new environment in which to translate the body of g. The new environment must contain the variable environment as it was when g was declared

$$\mathsf{b\_var}[\mathsf{b\_vb}(\mathsf{a}, \mathsf{id})]$$

and it should also incorporate the bindings from the environment produced by the formal parameter list:

$$\mathsf{b\_var}[\mathsf{b\_vb}(\mathsf{b}, \mathsf{id})]$$

The bindings for c and d must not appear, as they are not in the static scope of g, but the new environment must take account of the fact that two new variables have been declared since the function g was declared. The translator does this by inserting dummy bindings, giving us the new environment:

$$\mathsf{b\_var}[\mathsf{b\_vb}(\mathsf{b}, \mathsf{id}), \mathsf{b\_vb}(\mathsf{dummy}, \mathsf{id}), \mathsf{b\_vb}(\mathsf{dummy}, \mathsf{id}), \mathsf{b\_vb}(\mathsf{a}, \mathsf{id})]$$

The number of dummy values inserted is equal to the length of variable environment when the function is called minus the length of the variable environment when the function was declared. In this case the number of dummy entries is $3 - 1 = 2$.

The fourth step is the translation of the function body in the context of the new environment. This translation yields

$$+ \cdot (\mathsf{id} \cdot \pi_2, \mathsf{id} \cdot \pi_1 \cdot \pi_1 \cdot \pi_1)^\circ$$

where the first member of the alltup function is the translation of b and the second member is the translation of a.

$\boxed{B\_ENV \vdash EXP \Rightarrow B\_EXP}$

While expressions

$$\frac{\begin{array}{c} b\_env \vdash \texttt{fun\_app(}id_1, exp\texttt{)} \Rightarrow \mathsf{b\_comp}(b\_exp'', b\_exp) \\ b\_env \vdash \texttt{fun\_app(}id_2, exp\texttt{)} \Rightarrow \mathsf{b\_comp}(b\_exp', b\_exp) \end{array}}{b\_env \vdash \texttt{while(}id_1, id_2, exp\texttt{)} \Rightarrow \mathsf{b\_comp}(\mathsf{b\_while}(b\_exp'', b\_exp'), b\_exp)} \quad (16)$$

Figure 23. Translator rule for while expressions

The fifth and last step composes the translated code for the function body with the code for the actual parameter $(\mathsf{id}, \pi_2)^\circ$ to produce the final result of the function application:

$$+ \cdot (\mathsf{id} \cdot \pi_2, \mathsf{id} \cdot \pi_1 \cdot \pi_1 \cdot \pi_1)^\circ \cdot (\mathsf{id}, \pi_2)^\circ$$

When this code is combined with the code produced from the declaration sequence we have the translation of the entire program:

$$+ \cdot (\mathsf{id} \cdot \pi_2, \mathsf{id} \cdot \pi_1 \cdot \pi_1 \cdot \pi_1)^\circ \cdot (\mathsf{id}, \pi_2) \cdot (\mathsf{id}, + \cdot (\mathsf{id} \cdot \pi_2, 3)^\circ)^\circ \cdot (\mathsf{id}, 4)^\circ$$

The correctness of this translation can be verified:

$$\begin{array}{lrl}
+ \cdot (\mathsf{id} \cdot \pi_2, \mathsf{id} \cdot \pi_1 \cdot \pi_1 \cdot \pi_1)^\circ \cdot & & \\
(\mathsf{id}, \pi_2) \cdot (\mathsf{id}, + \cdot (\mathsf{id} \cdot \pi_2, 3)^\circ)^\circ \cdot (\mathsf{id}, 4)^\circ & val(\mathsf{a}) & = \\
+ \cdot (\mathsf{id} \cdot \pi_2, \mathsf{id} \cdot \pi_1 \cdot \pi_1 \cdot \pi_1)^\circ \cdot (\mathsf{id}, \pi_2) \cdot (\mathsf{id}, + \cdot (\mathsf{id} \cdot \pi_2, 3)^\circ)^\circ & (val(\mathsf{a}), 4) & = \\
+ \cdot (\mathsf{id} \cdot \pi_2, \mathsf{id} \cdot \pi_1 \cdot \pi_1 \cdot \pi_1)^\circ \cdot (\mathsf{id}, \pi_2) & ((val(\mathsf{a}), 4), 7) & = \\
+ \cdot (\mathsf{id} \cdot \pi_2, \mathsf{id} \cdot \pi_1 \cdot \pi_1 \cdot \pi_1)^\circ & (((val(\mathsf{a}), 4), 7), 7) & = \\
+ \, (val(\mathsf{a}), 7) & &
\end{array}$$

by checking that the original Adl program produces the same result, $+ \, (val(\mathsf{a}), 7)$, when applied to $val(\mathsf{a})$.

We have now described the bulk of the translator. The key mechanisms for producing code without variables have been established. Now we look at core functions whose translation depends on the mechanisms we have just described.

While expressions Figure 23 contains the rule for translating `while` expressions. The two premises derive the code for the two constituent functions and of `while`, along with the code for their, shared, parameter. These three sections of code are integrated into, and around, the BMF while function in the rule's conclusion.

$$\boxed{B\_ENV \vdash EXP \Rightarrow B\_EXP}$$

Map expressions

$$\frac{b\_env \vdash \texttt{fun\_app}(id, exp) \Rightarrow \mathsf{b\_comp}(b\_exp', b\_exp)}{b\_env \vdash \texttt{map}(id, exp) \Rightarrow \mathsf{b\_comp}(\mathsf{b\_comp}(\mathsf{b\_map}(b\_exp'), \mathsf{b\_op}(\mathsf{distl}()))), b\_exp)} \quad (17)$$

Figure 24. Translator rule for map expressions

This is the first time we have encountered the BMF while function in this report. The semantics of while are:

$$\begin{aligned} &\mathsf{while}(b\_exp_1, b\_exp_2)\ v \\ &\quad = v, \textit{\textbf{if}}\,(b\_exp_2\ v = true) \\ &\quad = \mathsf{while}(b\_exp_1, b\_exp_2)\ (b\_exp_1\ v), \textit{\textbf{otherwise}} \end{aligned}$$

Note that while is a partial function because an invocation of while can loop forever. Program transformations involving partial functions, such as while and div can change the termination behaviour of a program, especially in a strict setting[16].

Though such problems have not surfaced in tests thus far, systematic strategies for avoiding changes in termination behaviour need to be implemented.

Map expressions Rule 17 for translating map expressions is shown in figure 24. This rule translates Adl code of the form

```
map(g,e)
```

to BMF code of the form:

$$(< translation\_of\_g >) * \cdot \mathsf{distl} \cdot (\mathsf{id}, < translation\_of\_e >)^\circ$$

[16]Under strict evaluation, a transformation like:

$$\mathsf{if}(pred, \bot, alt) \Rightarrow \mathsf{if}(pred \cdot \pi_1, \pi_1 \cdot \pi_2, \pi_2 \cdot \pi_2) \cdot (\mathsf{id}, (\bot, alt)^\circ)^\circ$$

introduces non-termination. Likewise, the transformation:

$$\pi_2 \cdot (\bot, f)^\circ \Rightarrow f$$

removes non-termination. The introduction of non-termination can be avoided by not moving partial functions out of if functions. Avoiding the removal of non-termination is more problematic but not as pressing, though Mottl[104] gives a scenario where the removal of non-termination has serious implications.

We have not yet described the distl function[17]. distl distributes its left-hand argument over its right hand argument. An informal definition is:

$$\mathsf{distl}\,(v,[x_0,x_1,\ldots,x_{n-1}]) = [(v,x_0),(v,x_1),\ldots,(v,x_{n-1})]$$

The effect of distl can be observed by considering the translation of the Adl program

```
incr_by (a:vof int, delta:int) :=
  let
    g x = x + delta
  in
    map(g,a)
  endlet
```

to the BMF code:

$$(+\cdot(\pi_2,\pi_2\cdot\pi_1)^\circ)*\cdot\mathsf{distl}\cdot(\mathsf{id},\pi_1)^\circ$$

This code, applied to a pair of input values $([1,2,3,4],3)$, is evaluated:

$$\begin{array}{lrl}
(+\cdot(\pi_2,\pi_2\cdot\pi_1)^\circ)*\cdot\mathsf{distl}\cdot(\mathsf{id},\pi_1)^\circ & ([1,2,3,4],3) & = \\
(+\cdot(\pi_2,\pi_2\cdot\pi_1)^\circ)*\cdot\mathsf{distl} & (([1,2,3,4],3),[1,2,3,4]) & = \\
(+\cdot(\pi_2,\pi_2\cdot\pi_1)^\circ)* & \begin{array}{r}[(([1,2,3,4],3),1),(([1,2,3,4],3),2),\\ (([1,2,3,4],3),3),(([1,2,3,4],3),4)]\end{array} & = \\
(+)* & [(1,3),(2,3),(3,3),(4,3)] & = \\
[4,5,6,7] & &
\end{array}$$

Two observations, arising from this evaluation, are:

- Some distribution of data is required to ensure the second input value (3 in this case) is added to every element of the vector.
- The distribution of all input causes, unnecessarily, large intermediate data structures to be created.

The second of these two observations indicate that there is scope for using a more selective means of distribution. We explore this issue later in this chapter and describe ways of achieving more selective distribution in chapter 5.

[17] distl appears in the FP language described by Backus[13]. A related function distr or *distribute-right* appears in category theory as a defining natural transformation τ of strong functors [84].

As a final note, close scrutiny of rule 17 reveals an implementation trick. The rule applies the function inside the map to the *whole* vector rather than an element inside the input vector. This mis-match does not affect the correctness of the two sections of code produced by the rule for function application and avoids the need to define a new rule.

Reduce expressions In Adl, reduce expressions take the form:

`reduce(f,e,xs)`

where `f` is the name of a binary function, `xs` is an expression evaluating to an input vector and `e` is an expression representing the value returned by `reduce` when its input vector is empty. We often refer to `e` as the zero-element.

If the function `f` depends only on its parameters *and* the expression `e` contains no variable references, then a valid translation of `reduce` is:

$$\texttt{reduce(f,e,xs)} \Rightarrow < \mathit{translation\_of\_f} > /_{<\mathit{translation\_of\_e}>} \cdot < \mathit{translation\_of\_xs} >$$

This translation schema applies in most circumstances. Unfortunately, in the rare cases where `e` accesses variables, the translation of `reduce` must produce code that transports values, referenced by these variables, to the translation of `e`. Compounding this problem, are cases where the binary function `f` contains references to values other than its parameters, that is, it references free variables. To support such references the translator must produce code to distribute values of these variables over each element of the input vector and also supply code to help the translation of `f` extract values from its parameters in the way it expects.

The combination of code required to support variable references in `e` and free variable references in `f` is quite complex. Such references can legitimately occur in Adl source code and the translator, being conservative, has a rule for `reduce` that produces this complex code. Fortunately, this complex code is, except in rare cases, short-lived in our implementation. The optimisation stage of our compiler, described in chapter 5 ascertains the references made by translator code and simplifies the translator's BMF code accordingly. The practical effect is, that for most instances of `reduce`, the simplified translation above is the one that applies.

Here, we do not show the complex, conservative, translator rule for `reduce`. A full description of the rules for `reduce`, `scan` and all of their variations is given in[6].

reducep, reducel, reducer, reducelp, reducerp, and all varieties of scan The translator has separate rules for all variants of `reduce` and `scan`. All of these rules are too similar to the rule for `reduce` to warrant a separate description. The main difference is that rules for `reducep`, `reducelp`, `reducerp`, `scan`, `scanl`, and `scanr` are less complex because they do not handle a zero-argument.

This concludes our detailed description of the rules of the translator. We now examine the performance of translator code.

4.4 Performance

In this section we analyse, by way of examples, the performance of the code produced by the translator. Prior to any discussion of performance, we need to establish a metric for sequential performance.

4.4.1 Measuring sequential performance

Measurement of the translator's performance needs both an execution model and baseline of BMF programs against which to compare the performance of the translator's programs. Our model though very simple, incorporating none of the vagaries involved with running programs on modern architectures[18], is sufficient for the gross comparsions made in this context.

4.4.1.1 Execution model for measuring sequential performance

A semi-formal description of the sequential execution model used in this chapter and the next is given in appendix D. To avoid the need to examine this code in detail we briefly characterise the model's behaviour by stating the conventions it consistently applies.

Conventions of the sequential model The following conventions apply to the sequential performance model:

- Storage for output of each scalar function is allocated just prior to evaluation of that function.

[18]Just some of these are measured and discussed in appendix F.

- Storage for input data of each scalar function is de-allocated just after evaluation of each function.
- The time taken to allocate or deallocate any amount of storage is one time unit.
- With the above in mind, functions on scalar data, such as $+$ and $\times$ have the following execution profile:
  1. allocate storage for the result (one time unit).
  2. perform the operation (one time unit).
  3. deallocate storage for the input (one time unit).
- A vector takes up one unit of space plus the combined space used by its elements.
- A tuple takes up a space equal to the combined space used by its elements.
- The cost of copying one item of scalar data is one time unit.
- During aggregate operations, data is copied/allocated at the last possible moment and deallocated at the first possible moment.

Using the design decisions above it is straightforward to derive an execution model for each BMF primitive. Some examples follow.

Execution models for some functions The following descriptions of execution models for a selection of functions may assist in the interpretation of the examples that follow.

length The length function, although it works on vectors, takes only three time units. Its operation is as follows:

1. Allocate memory for the scalar result (one time unit).
2. Perform the length function by looking at the vector descriptor (one time unit).
3. Deallocate the storage allocated to the input vector (one time unit).

The length function has a peak storage consumption of one plus the storage taken up by the vector. Note, the index function has very similar characteristics to the length function.

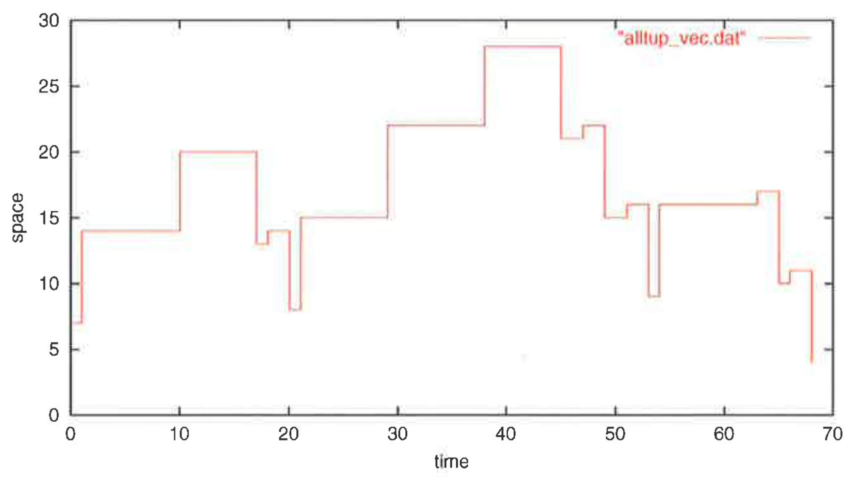


Figure 25. Time/Space graph of sequential evaluation of the expression: $(\# \cdot \pi_1, !.(\pi_1, \pi_2)^\circ, \pi_2, 3)^\circ\ (3, [1, 2, 3, 4, 5])$

iota The iota function operates as follows:

1. Allocate the result vector (one time unit).

2. Fill up the vector (time equivalent to the number of items in the vector).

3. Deallocate the input value (one time unit).

Note that the model does not currently account for the very small amount of time taken to compute the next value in the result vector.

alltup alltup functions create a copy of the input value, in order to apply it to each component of the alltup. The copy is made just prior to the application of each component. There is no copy made just before the last component is applied. The trace of the expression:

$$(\# \cdot \pi_1, !.(\pi_1, \pi_2)^\circ, \pi_2, 3)^\circ\ (3, [1, 2, 3, 4, 5])$$

shown in the graph in figure 25 illustrates the behaviour of alltup. The significant features of the trace are the copying of the input values for the first, second, and third components of the alltup function starting at times 0, 20, and 52 respectively. On each occasion, copying takes time equal to the length of the value being copied. Copying takes a large proportion of the processing time for the alltup expression[19]

[19]Much of this copying time can be eliminated by preventing all but the last function in the alltup from deleting its input value. If we maintain this "no-delete-unless-I'm-last" policy we could get

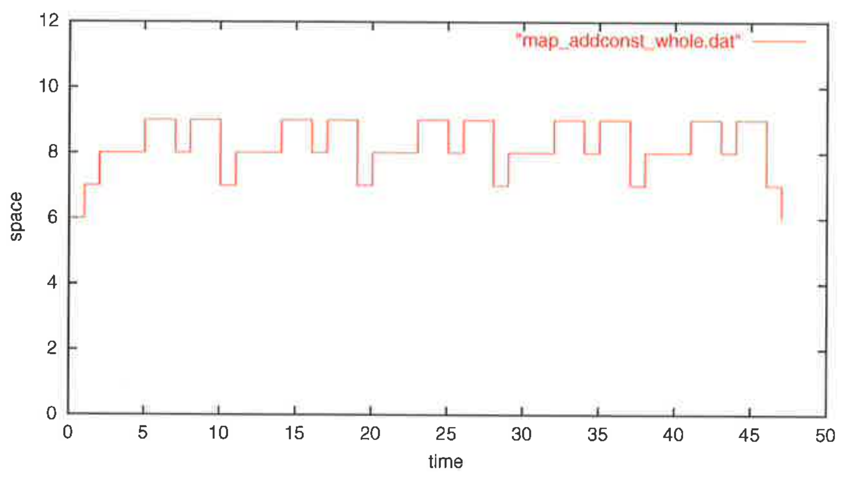


Figure 26. Time/Space graph of sequential evaluation of the expression: $(+\cdot(1, id)^{\circ}) * \; [1, 2, 3, 4, 5]$

map The map function allocates an empty output vector, taking up one unit of space, and then consumes elements of the input vector, one at a time, allocating space for output values and deallocating input values as it proceeds. This behaviour gives map a quickly-alternating pattern of allocation and deallocation, evident in the trace of the expression:

$$(+\cdot(1, id)^{\circ}) * \; [1, 2, 3, 4, 5]$$

shown in figure 26.

reduce Reduce evaluates its zero-argument before using its binary function to perform a sequential reduction of the vector. In our model, evaluation of the zero-argument requires a copy of the input vector, just in case the evaluation of the zero-argument demands it.

The production of the result consumes the input vector one element at a time. Figure 27 shows a trace of the evaluation of the expression:

$$+/_0 \, [1, 2, 3, 4, 5]$$

The decline of space consumption in the latter half of the trace represents the gradual consumption of the input vector.

by with one copy of the input. Unfortunately, this policy complicates the model beyond what is necessary for gross comparisons of program efficiency we need here.

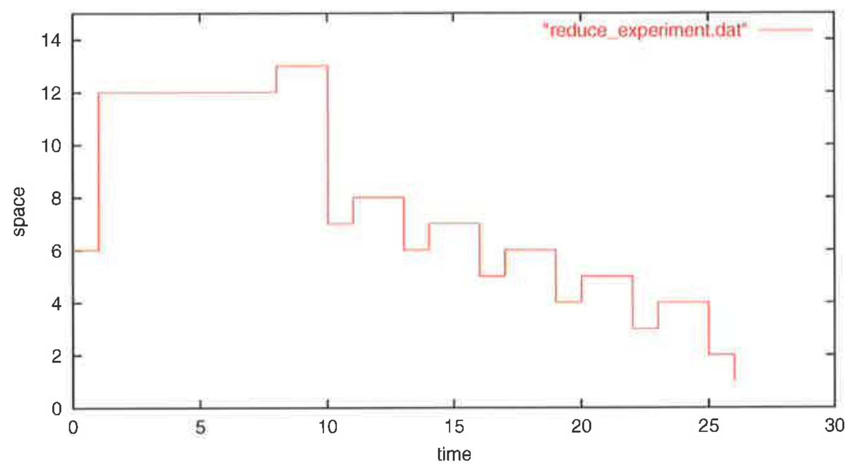


Figure 27. Time/Space graph of sequential evaluation of the expression: $+/_0\ [1, 2, 3, 4, 5]$

The examples above, in combination with the conventions outlined earlier, illustrate and delineate our metric for sequential performance. Now, we need to establish a benchmark for comparison of translator code performance.

4.4.1.2 Baseline for comparison

We use hand-crafted BMF programs as benchmarks for assessing efficiency of translator and, in the next chapter, optimiser code. These benchmark programs are built with the aim of obtaining a fast execution time without excessive use of memory[20]. Now, we perform a series of comparisons between translator code and our hand-crafted programs.

4.4.2 Examples of performance of translator code

We present a sample of problems whose solutions demonstrate the performance characteristics of translator code.

4.4.2.1 Adding a constant to elements of a nested vector

An Adl program, called `map_map_addconst.Adl`, that adds a constant to elements of a nested input vector is shown in figure 28

[20]Using an excessive amount of memory tends to slow down BMF programs in any case; in BMF operations have to copy data around themselves and that movement takes time.

```
main a:vof vof int :=
    let
       f x :=
          let
             g y := y + 2
          in
             map (g,x)
          endlet
    in
       map (f,a)
    endlet
```

Figure 28. `map_map_addconst.Adl`, an Adl program that adds a constant to each element of a nested vector

$$((+\cdot(\pi_2,2)^\circ)*\cdot\mathsf{distl}\cdot(\mathsf{id},\pi_2)^\circ\cdot\mathsf{id})*\cdot\mathsf{distl}\cdot(\mathsf{id},\mathsf{id})^\circ\cdot\mathsf{id}$$

Figure 29. BMF code, for `map_map_addconst.Adl`, produced by the translator.

The translation of `map_map_addconst.Adl` is shown in figure 29. The translation also consists of nested calls to map functions. The distl functions ensure that each invocation of each map body has access to all of the elements of the nested input vector. Such broad access is not required but it is not the translator's job to determine this.

Figure 30 shows an equivalent hand-crafted BMF program. The relative performance of these two programs when presented with the data:

$$[[1,2,3],[4,5,6],[7,8,9]]$$

is shown in figure 31. The top line shows the time and space consumption of the code produced by the translator and the bottom line shows the time and space consumption of the hand-crafted BMF code. The very large gap in performance means there is considerable scope for optimisation. The next chapter outline an optimisation process

$$((+\cdot(\mathsf{id},2)^\circ)*)*$$

Figure 30. Hand-crafted BMF code for `map_map_addconst.Adl`

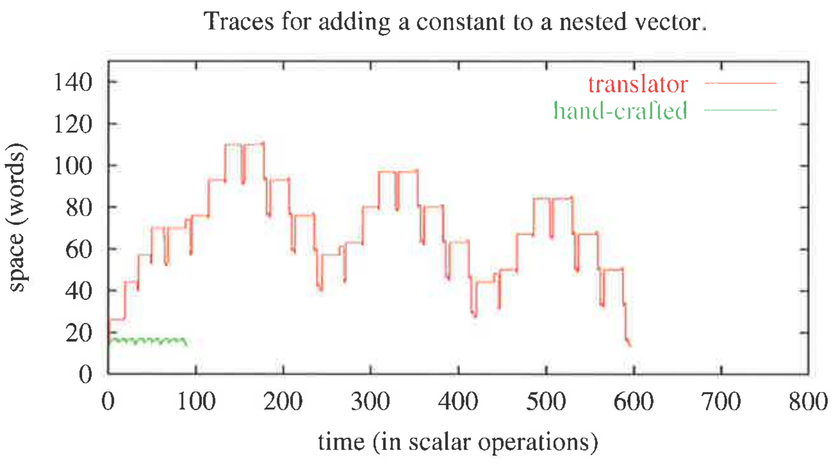


Figure 31. Time/Space graph of sequential evaluation of translator and hand-crafted BMF code for `map_map_addconst.Adl` applied to the nested vector $[[1,2,3],[4,5,6],[7,8,9]]$.

to bridge most of this gap. For now, we will focus on the reasons for this performance gap.

The translator code suffers from the overhead of executing nested distl functions. The most upstream distl function is responsible for the first part of the initial rise in the top-most curve. The rising steps in each of the three peaks that follow are caused by the application of the nested distl further downstream. Both the translator and hand-crafted code produce the same result but the translator version uses a lot more time and space in the process. Moreover, this performance gap increases rapidly with both the size and level of nesting of the input vector. This is because the translator produces code whereby all values in scope are distributed over each vector in each map operation. At the very least, the values in scope include the vector itself, resulting in code the space consumption of which grows, at least, with the square of the size of the input vector to the map operation.

4.4.2.2 Summing a list of numbers (using reduce)

The program `sum.Adl`, shown in figure 32, uses `reduce` to produce a sum of the numbers in its input vector.

Figure 33 shows the equivalent translator, and handcrafted code respectively. As mentioned in section 4.3.3.2, the translation of `reduce` is very conservative. The translator assumes access to any value in scope *might* be required as the reduction

```
main a: vof int :=
let
   add(x,y) := x + y
in
  reduce (add,0,a)
endlet
```

Figure 32. `sum.Adl`, a program to sum a vector of integers

$$\mathsf{if}(\neq \cdot (0, \# \cdot \pi_2)^\circ, \pi_2 \cdot ((\pi_1 \cdot \pi_1, + \cdot (\pi_1 \cdot \pi_1, (\pi_2 \cdot \pi_1, \pi_2 \cdot \pi_2)^\circ)^\circ)^\circ)/ \cdot \mathsf{distl}, \mathsf{0} \cdot \pi_1) \cdot (\mathsf{id}, \mathsf{id})^\circ$$

(a)

$$+/_0$$

(b)

Figure 33. Translator code (part(a)) and hand-crafted code (part(b)) for `sum.Adl`.

proceeds. In reality, the need for such access rarely arises, the translator produces quite detailed code to support it nonetheless. A full explanation of this code is given in[6]. In contrast to translator code, handwritten BMF version of `sum.Adl` is very simple:

$$+/_0$$

We have already analysed the performance of this code on page 78. Figure 34 compares the performance of translator code to hand-crafted code. The very large performance gap, already evident with very small data, shows there is much scope for optimisation. Fortunately, as we shall see in the next chapter, the optimiser works very effectively on most instances of reduce.

What about scan? The observations we have just made regarding reduce also apply to scan. Both have an inefficient initial translation, both have an efficient hand-crafted implementation and both translated versions are amenable to optimisation. Next, we give an example where crafting an efficient solution requires deeper analysis.

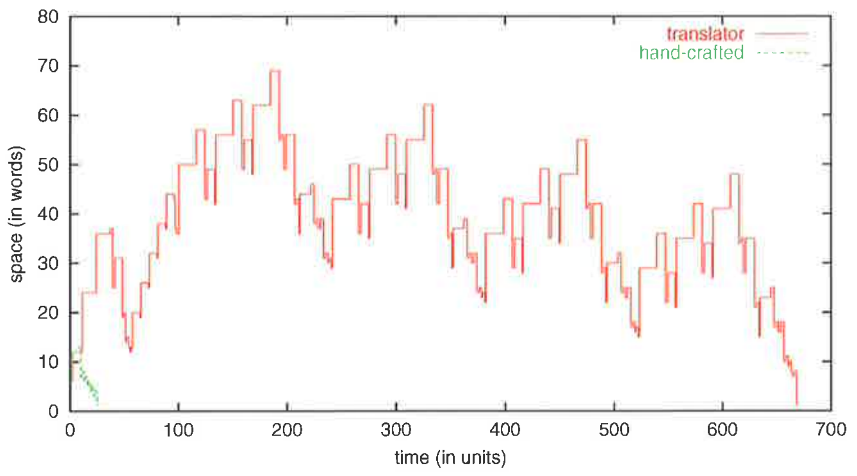


Figure 34. Time/Space graph of sequential evaluation of translator and hand-crafted BMF code for `reduce_plus.Adl` applied to the vector $[1,2,3,4,5]$.

4.4.2.3 A one-dimensional finite-difference computation

Finite difference techniques are commonly used in simulations of physical systems and in image processing [57]. Finite difference computations are performed on arrays of values. Each element of the array has a value or set of values. Values in each elements are combined in some way with those of neighbouring elements to produce a new result. This localised interaction at each datum can be applied repeatedly until the goal of the computation is reached. The localised nature of operations in finite difference algorithms makes them very well suited to parallel execution.

The Adl program in figure 35 contains the basic elements of a finite-difference application. The array in this case is an un-nested vector. A new value for each element is derived from the element's own value and the values on either side. The function for deriving a new value is called `stencil`[21]. To simplify our example we apply `stencil` only once for each vector element and the boundary elements of the vector are not included in the result. The translation of `finite_diff.Adl` is shown in figure 36 and the hand-crafted BMF version of `finite_diff.Adl` is shown in figure 37. There are only minor differences between the translator and hand-crafted code for `finite_diff.Adl`. Both versions of code distribute the entire input vector to each invocation of the `map` function that produces the result. These slight differences do have an impact on efficiency, a fact that is borne out by the relative performances of the two programs as shown in figure 38. However, the

[21]Stencil is the standard term for the pattern of interaction at each element in these computations.

```
main a: vof int :=
  let
    stencil x := a!x + a!(x-1) + a!(x+1);
    addone x := x + 1;
    element_index := map(addone,iota ((# a)-2))
  in
    map (stencil, element_index)
  endlet
```

Figure 35. `finite_diff.Adl`, an Adl program to perform a simple finite-difference computation.

$$(+ \cdot (+ \cdot (! \cdot (\pi_1 \cdot \pi_1, \pi_2)^\circ, ! \cdot (\pi_1 \cdot \pi_1, - \cdot (\pi_2, 1)^\circ)^\circ)^\circ, ! \cdot (\pi_1 \cdot \pi_1, + \cdot (\pi_2, 1)^\circ)^\circ)^\circ) * \cdot$$
$$\mathsf{distl} \cdot (\mathsf{id}, \pi_2)^\circ \cdot \mathsf{id} \cdot$$
$$(\mathsf{id}, (+ \cdot (\pi_2, 1)^\circ) * \mathsf{distl} \cdot (\mathsf{id}, \mathsf{iota} \cdot - \cdot (\# \cdot \mathsf{id}, 2)^\circ)^\circ)^\circ$$

Figure 36. Code from automated translation of `finite_diff.Adl` into BMF

$$(+ \cdot (+ \cdot (! \cdot (\pi_1, \pi_2)^\circ, ! \cdot (\pi_1, - \cdot (\pi_2, 1)^\circ)^\circ)^\circ, ! \cdot (\pi_1, + \cdot (\pi_2, 1)^\circ)^\circ)^\circ) * \cdot$$
$$\mathsf{distl} \cdot$$
$$(\mathsf{id}, (+ \cdot (\mathsf{id}, 1)^\circ) * \cdot \mathsf{iota} \cdot - \cdot (\# \cdot \mathsf{id}, 2)^\circ)^\circ$$

Figure 37. Hand-coded translation of `finite_diff.Adl` into BMF

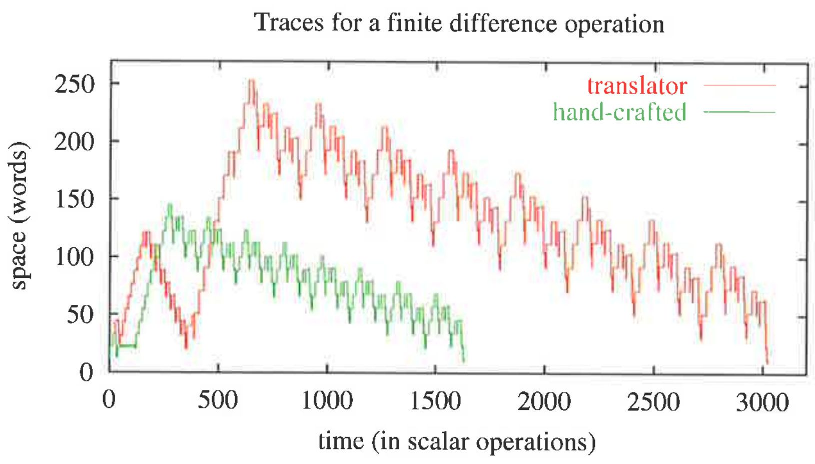


Figure 38. Time/Space graph of sequential evaluation of translator and hand-crafted BMF code for `finite_diff.Adl` applied to the vector $[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$.

$$\begin{array}{l}
(+ \cdot (\pi_1 \cdot \pi_1, + \cdot (\pi_2 \cdot \pi_1, \pi_2)^\circ)^\circ) * \cdot \\
\mathsf{zip} \cdot (\mathsf{zip} \cdot (\pi_1 \cdot \pi_1, \pi_2 \cdot \pi_1)^\circ, \pi_2)^\circ \cdot \\
((\mathsf{select}, \mathsf{select} \cdot (\pi_1, (+ \cdot (\mathsf{id}, 1)^\circ) * \cdot \pi_2)^\circ)^\circ, \mathsf{select} \cdot (\pi_1, (- \cdot (\mathsf{id}, 1)^\circ) * \cdot \pi_2)^\circ)^\circ \cdot \\
(\mathsf{id}, (+ \cdot (\mathsf{id}, 1)^\circ) * \cdot \mathsf{iota} \cdot - \cdot (\#, 2)^\circ)^\circ
\end{array}$$

Figure 39. A more efficient version of `finite_diff.Adl` using new functions.

performance of neither version of code is impressive. Even the hand-coded version takes over 1600 operations to perform less than 30 additions. More significantly, as we shall soon see, the performance of both versions of code gets steadily worse as the problem size grows. Both BMF versions of `finite_diff.Adl` waste time and space distributing the entire input vector to each element; we only need three elements for each invocation of map not the whole vector. This waste is unavoidable if we restrict ourselves to using the BMF functions we have seen thus far. That is, the hand-crafted version of `finite_diff.Adl` is close to optimal given the limited set of tools we have. Fortunately, as we shall see, we can improve matters if we add more BMF primitives to our lexicon.

Figure 39 shows a more efficient translation of `finite_diff.Adl` using extra primitives. The functions in question are zip, from the previous chapter, and, more importantly, a new function select. The zip function converts a pair of vectors into a vector of pairs. The select function takes a pair consisting of a source vector and an index vector. select produces a vector consisting of the elements of the source vector that are indexed by the elements of the index vector. For example:

$$\mathsf{select}\ ([10, 20, 30, 40, 50], [0, 3, 4, 3, 0, 1])$$

produces:

$$[10, 40, 50, 40, 10, 20]$$

The exact details of the code in figure 39 are not important at this point, and we will revisit similar examples in detail in chapter 5; the important observations to make are that:

- a means exists, to make access to elements of vectors from inside map efficient in BMF and that
- to exploit this means we are forced use more BMF primitive functions and make *significant* alterations to the structure of the program.

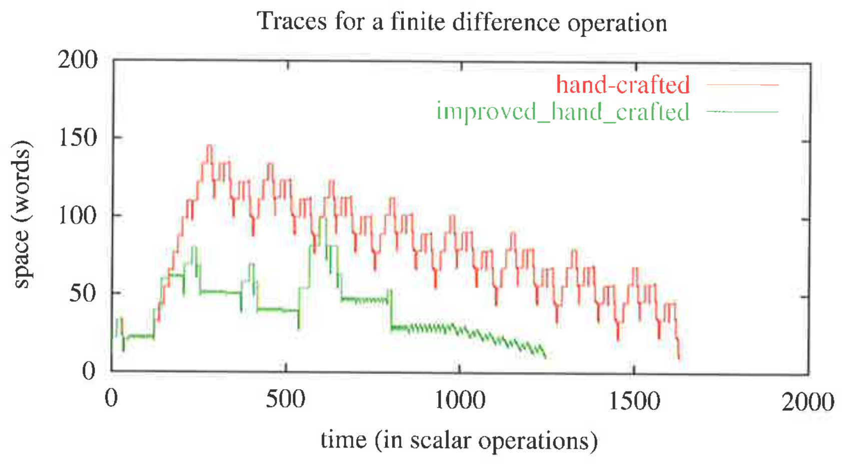


Figure 40. Time/Space graph comparing sequential evaluation of hand-crafted BMF code for `finite_diff.Adl`, using existing functions to hand-crafted code using the zip and select functions. The input data used is the vector $[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$. Note the change of scale from figure 38.

The extent to which efficiency is improved over the earlier hand-crafted version is illustrated by figure 40. Note that the scale has changed to enlarge the two traces of interest. The gap in performance between each of the three versions of BMF code for `finite_diff.Adl` grows quickly with the size of the input data. Figure 41 compares the BMF code produced by the translator with hand-crafted BMF code and with the improved hand-crafted BMF code. There is a significant gap between the top-line and the middle-line. This gap represents the improvement to be had by eliminating transmission of superfluous members of tuples. There is an even more significant gap between the middle line and the bottom line. This gap represents the improvement to be had by being selective about transmission of individual vector elements. The bottom line is linear, which means it the has the same time complexity as the best solutions to this problem[22]. In contrast, the two upper curves are quadratic and sub-optimal. This difference is very important, it tells us that if we want to arrive at a BMF solution that has an optimal order of efficiency we have to be prepared to perform optimisation that exploits patterns of vector access and, in the process, makes substantial changes the structure of the translator code. This is the work of the optimiser and is described in the next chapter. A short summary of related work is given next.

[22]The best sequential solutions to a one-dimensional finite-difference problem of this sort have linear time complexity.

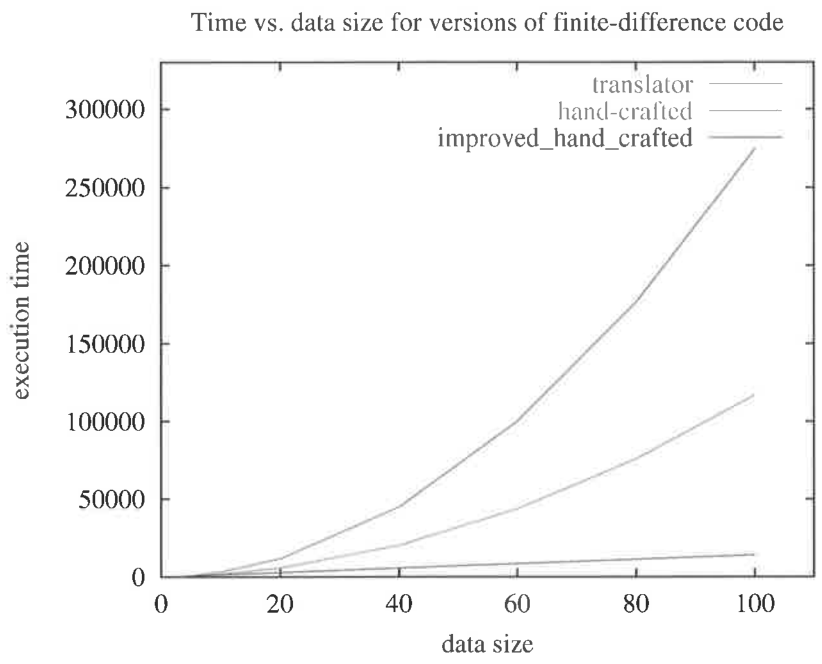


Figure 41. The relative performance of different BMF versions of `finite_diff.Adl` applied to different sized data sets. As before, execution time is measured in scalar operations.

4.5 Related Work

To the best of our knowledge closest relative to the Adl to BMF translator is the EL* to FP* translator created by Banerjee and Walinsky[117]. EL*, like Adl, is a small strict functional language aimed toward the expression of of data parallel algorithms. Both Adl and EL* support vector and tuple aggregate types and support selection on these types.

Adl provides primitive operations such as `map`, `reduce` and `scan` for specifying operations over vector aggregates and `iota` for allocation of new aggregates. In contrast, EL* allows the use of a restricted form of recursion to express divide and conquer parallelism. These recursive functions are mapped into a standard FP* template consisting of code to align and distribute input before performing a `map` on the divided data before using a `reduce` to combine expressions. Another point of contrast is the handling of references to free variables within functions passed as parameters. Such references are supported in Adl with the help of function closures but they are not supported in EL*.

Some translation techniques are similar between the two implementations. For example, the mechanism translating the passing of the referencing environment through declaration sequences appears similar. Both translators also inject excess distribution of data into the code, though the Adl translator seems to do this to a larger extent due to the richer referencing environment allowed for functional arguments.

In earlier work, the translation process, described in[100], from a simple functional language to the Catagorical Abstract Machine (CAM) also maps a pointwise functional language to an almost point-free form. The primary difference between the CAM translation rules and the Adl translator is the use of a stack in CAM. This stack does not have to be explicitly transported by code as the referencing environment is in Adl translator code. However, the stack is accessed in a similar manner in CAM code to the way the nested input tuples are accessed in translated Adl code.

A larger and more recent body of related work defines rules to assist the conversion of recursive functions into compositional form (BMF). Hu[78] describes a partly mechanised process for converting single recursive definitions in a functional language into BMF. This is a very much more open-ended task than that attempted by the Adl to BMF translator. The motivation behind this work appears similar to the motivation for our translator: once a program is expressed in compositional form (BMF) it is more amenable to transformation and parallelisation.

The thrust of the above work is extended by Cunha et. al[39] where a translation is defined between pointwise Haskell and a point-free functional form. Their work differs from this translation in its scope which covers the translation of recursive and higher-order functions in a lazy language. The Adl translator has more emphasis on translation of numerical and array operations and works in the context of a strict language. Again, the methodology for handling the environment in both translations is similar, using nested tuples, though the Adl translator supports a slightly richer variety of tuple types.

This concludes our brief description of related work. The conclusions drawn from this chapter are presented next.

4.6 Conclusions

From the work carried out to create this chapter we arrive at the following conclusions.

Feasibility of translation Direct translation from Adl, a simple functional language, to BMF a notation without variable references, is feasible and has been implemented. Our implementation is an executable semantic model of the translation process. Our implementation is conservative, it creates BMF code where all the values that were accessible to functions in the Adl source are accessible to corresponding functions in the BMF translation. The translator's conservative nature keeps the process mechanistic, deep analyses are avoided, but the code produced is not efficient.

Efficiency of translated code The translator converts from a notation where values can be kept in scope at little cost (Adl) to a notation where keeping values in scope incurs the cost of copying and transporting them. This cost is substantial where functions are applied to each element of a large data set, as can occur in functions like map.

What optimised code looks like It is clear that some optimisation of translator code is required but what will be required of this optimisation process? We perceive there are two parts. First, we want to eliminate the transport of surplus components of tuples. Second, we want to eliminate the transport of surplus components of vectors. The second form of optimisation requires the introduction of new primitives and substantial changes to program structure.

Summary In this chapter we have shown that a translator from Adl to BMF can be built and how such a translator is constructed. We looked at the code produced by the translator and showed that this code needs to be optimised. Lastly, we looked at what would be required of an optimisation process.

In the next chapter, we explore an automated optimisation process for the BMF code produced by the translator.

Chapter 5

Data Movement Optimisation

The last chapter described a translator from Adl to BMF. This translator, whilst reliable, was very conservative in choosing values to transmit through the program. In this chapter we describe an optimiser that makes code more selective in the values it chooses to transmit and, by this process, greatly increases the efficiency of code.

5.1 Chapter outline

The role of the optimiser, in the context of our implementation, is shown in figure 42 As can be seen, the optimiser takes sequential BMF code from the translator and produces optimised sequential BMF code.

5.1.1 How the optimiser is described

The core of our prototype implementation of the optimiser, like the translator, is defined using rules expressed in Natural Semantics.

This optimiser definition consists of several hundred rules, too many to present here. Instead we focus on the key elements of the optimisation process and, wherever possible, keep the discussion informal and provide illustrative examples.

5.1.2 Stages of optimisation

The results, shown at the end of the last chapter, highlighted two factors that make translator code inefficient:

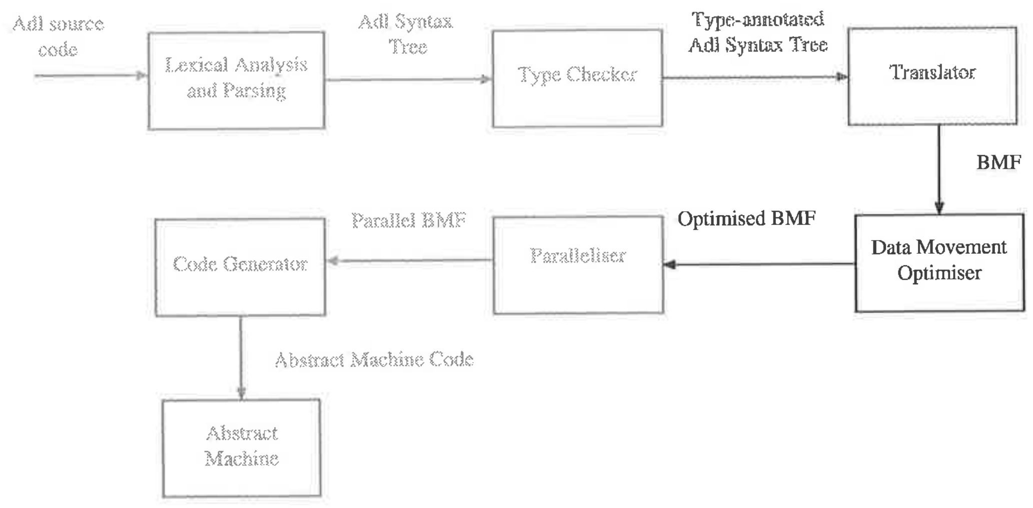


Figure 42. The Adl project with the optimisation stage highlighted in black.

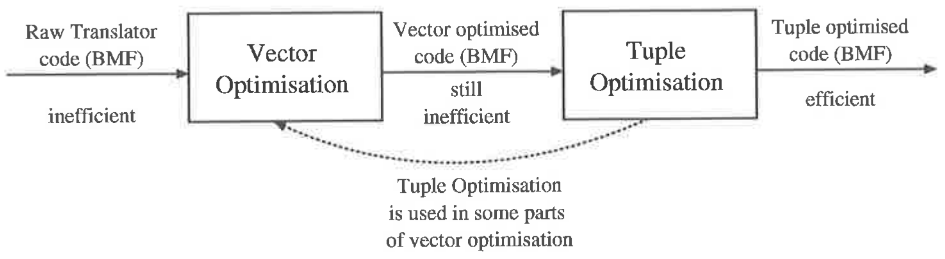

Raw Translator code (BMF)
inefficient
Vector Optimisation
Vector optimised code (BMF)
still inefficient
Tuple Optimisation
Tuple optimised code (BMF)
efficient
Tuple Optimisation is used in some parts of vector optimisation

Figure 43. The stages of the optimisation process

1. Entire vectors are copied to each instance of a function inside map functions, even when only a small portion of the vector is required by each instance.

2. Members of tuples *not* required by functions downstream are, nonetheless, transported to functions downstream.

The first factor is remedied by altering code so that vector elements are directed only to the places that need them. We call this remedy *vector optimisation.* The second problem is remedied by altering code so that elements of tuples that will not be needed, are not passed downstream. We call this remedy *tuple optimisation.*

In the Adl implementation, tuple optimisation and vector optimisation are performed as two, almost, separate passes. Figure 43 shows these stages of optimisation. As can be seen, vector optimisation is performed first, followed by tuple optimisation. Both stages produce and consume BMF code [1]. The vector optimisation stage invokes tuple optimisation on some small sections of code but

[1]Vector optimisation produces code containing new primitive functions but the code is still BMF.

tuple optimisation does not invoke vector optimisation. Efficient code is produced when the last stage of optimisation is complete.

5.1.3 Common elements and separate elements

There are many differences between vector optimisation and tuple optimisation but these processes also share common elements. The next two sections outline the *common strategy* and *common tactics* respectively. Sections 5.4 and 5.5 describe the distinct elements of vector and tuple optimisation. Section 5.6 assesses the performance of the optimiser and the code it produces and, finally, section 5.7 surveys related work and concludes the chapter.

5.2 The common strategy

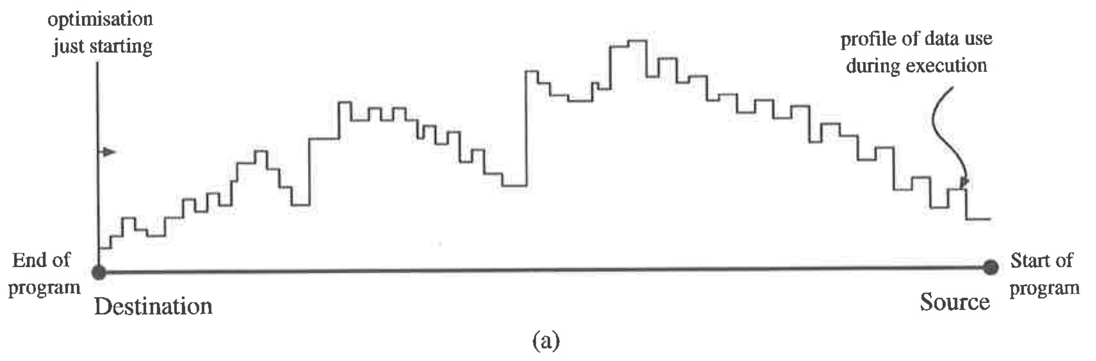
Both vector and tuple optimisation share the following strategy:

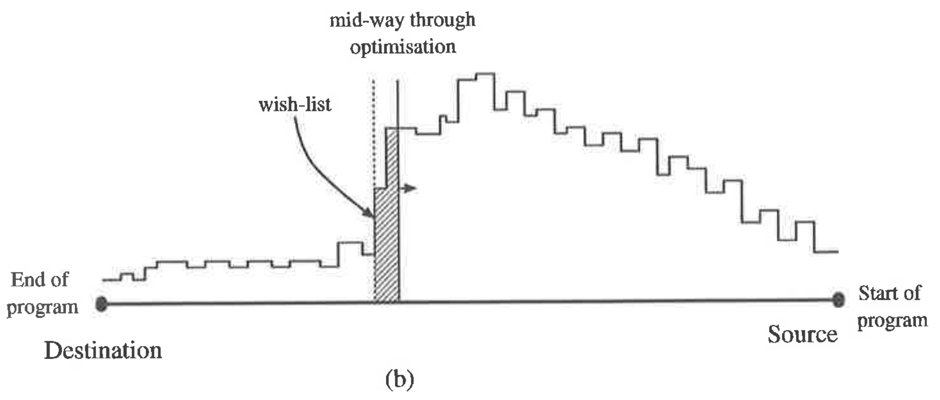
> Minimising the amount of data produced at each source of data so it more precisely matches the needs of the destination of the data.

Destination and source Where, in a BMF program, is the destination and source? In BMF, data starts on the right, in the most upstream parts of the program, and proceeds to the left to the innermost, most-downstream parts.

The path of optimisation Both vector and tuple optimisation start at the most downstream part of the program and push the optimisation process, outwards and upstream toward the source of data. Figure 44 provides a conceptual illustration of this process and its effect on both data usage and elapsed execution time. The three parts of the figure show snapshots of performance profiles of code at three stages of an optimisation pass, vector or tuple[2]. In part (a) the optimisation pass has not yet started. If the code were executed at this stage it would, typically, use a lot of data, and take a long time to run. Part (b) is a snapshot of code half-way through optimisation. At this stage the optimiser has reduced the data usage and execution time of the downstream half of the program. There is a section of transitional code,

[2]The diagram most resembles the progress of tuple optimisation because it is the last pass where all of the benefits of both stages of optimisation are made manifest.





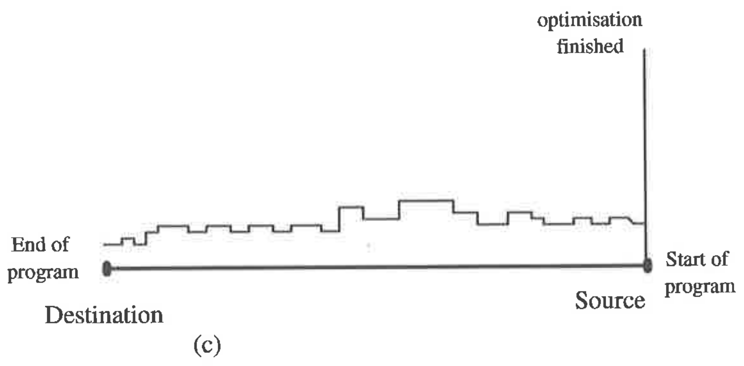


Figure 44. The effect of optimisation on program code from the start of optimisation, part (a), to the mid-way through optimisation, part (b), to the final result of optimisation part (c).

called the *wish-list*, that expresses the data needs of downstream code to upstream code, the wish-list will be more fully described shortly. Part (c) of figure 44 gives a profile of the fully optimised program, which uses substantially less data and takes substantially less time to run than the unoptimised program.

Note that both tuple and vector optimisation rely heavily on the wish-list always providing an accurate expression of the data needs of the code downstream of it. We describe what the wish-list is, and how it encodes data needs next.

The wish-list: encoding data needs The wish-list is encoded slightly differently for vector and tuple optimisation:

- In vector optimisation, the wish-list consists entirely of ordinary BMF code.
- In tuple optimisation, the wish-list is injected into a filter expression whose contents is either BMF code or a special value, called null.

We will investigate the filter expression and the role of null further in our description of in section 5.5. At this point, it is sufficient to note that a filter expression containing null can be safely treated as if it is the BMF id function. The main point is that, in all cases, the wish-list either entirely consists of BMF code or can be easily mapped to BMF code.

Encoding the wish-list as code avoids the need for extra syntax and rules to handle a special form for wish-lists. It also means that the infrastructure used to guide optimisation, the wish-list, is also part of the program itself. Optimisation could be stopped at any point where a new wish-list is produced and, with very little effort, a correct program forming a snapshot of that point of the optimisation process can be extracted. The longer optimisation is allowed to run the more efficient these snapshots get. The process is one of incremental convergence towards an efficient solution[3]. Furthermore, each snapshot provides extra information that can be gathered by the implementor for the process of fine-tuning and debugging the implementation.

Figure 45 illustrates the use of a wish-list, as applied to tuple optimisation. Part (a) shows a simple program prior to optimisation. Part (b) shows the initial phases

[3]It is worth noting that this convergence is not strictly monotonic. Auxiliary rule-sets that perform normalisation of code in preparation for future optimisation steps can, temporarily, decrease the efficiency of code. Moreover, each of the two phases of optimisation is focused on one aspect of performance and thus significantly improved performance may not be manifest until both phases are complete.

$$(+ \cdot (\pi_1 \cdot \pi_2, \pi_2 \cdot \pi_2)^\circ, + \cdot (\pi_1 \cdot \pi_2, 3)^\circ)^\circ \cdot (\mathsf{id}, (\pi_2, \pi_1)^\circ)^\circ \cdot (\mathsf{id}, 2)^\circ$$

(a)

$$\begin{array}{lll}
 & (+ \cdot (\pi_1 \cdot \pi_2, \pi_2 \cdot \pi_2)^\circ, + \cdot (\pi_1 \cdot \pi_2, 3)^\circ)^\circ \cdot (\mathsf{id}, (\pi_2, \pi_1)^\circ)^\circ \cdot (\mathsf{id}, 2)^\circ & (1) \\
\Rightarrow & (+ \cdot (\pi_1, \pi_2)^\circ \cdot \underline{(\pi_1 \cdot \pi_2, \pi_2 \cdot \pi_2)^\circ}, + \cdot (\mathsf{id}, 3)^\circ \cdot \underline{\pi_1 \cdot \pi_2})^\circ \cdot (\mathsf{id}, (\pi_2, \pi_1)^\circ)^\circ \cdot (\mathsf{id}, 2)^\circ & (2) \\
\Rightarrow & (+ \cdot (\pi_1, \pi_2)^\circ, + \cdot (\mathsf{id}, 3)^\circ \cdot \pi_1)^\circ \cdot \underline{(\pi_1 \cdot \pi_2, \pi_2 \cdot \pi_2)^\circ} \cdot (\mathsf{id}, (\pi_2, \pi_1)^\circ)^\circ \cdot (\mathsf{id}, 2)^\circ & (3) \\
\Rightarrow & (+ \cdot (\pi_1, \pi_2)^\circ, + \cdot (\mathsf{id}, 3)^\circ \cdot \pi_1)^\circ \cdot (\pi_1, \pi_2) \cdot \underline{(\pi_2, \pi_1)^\circ} \cdot (\mathsf{id}, 2)^\circ & (4) \\
\Rightarrow & (+ \cdot (\pi_1, \pi_2)^\circ, + \cdot (\mathsf{id}, 3)^\circ \cdot \pi_1)^\circ \cdot (\pi_1, \pi_2) \cdot (2, \mathsf{id})^\circ & (5)
\end{array}$$

(b)

Figure 45. Part (a), an unoptimised program. Part (b), steps in the optimisation process with the wish-list underlined.

of tuple-optimisation applied to the program (more on this soon). The wish-list is underlined The wish-list is so-named because it precisely defines the data needed by the downstream code. That is, it expresses the wishes of the code downstream. Almost invariably, in unoptimised code, the downstream code wishes for a lot less data than it actually gets.

Sweeping upwards and outwards The basic strategy used by tuple optimisation (but also used by vector optimisation) can be seen in part (b) of figure 45. Firstly, the most downstream parts are modified to become what they would be if given precisely the data required to produce their output. So, for example, the leftmost expression on line (1) of part (b) is converted from:

$$+ \cdot (\pi_1 \cdot \pi_2, \pi_2 \cdot \pi_2)^\circ$$

to

$$+ \cdot (\pi_1, \pi_2)^\circ$$

Simultaneously, a wish-list is inserted to provide the modified downstream code the data it now requires. The simultaneous modification of downstream code and insertion of a wish-list preserves program semantics. This preservation of semantics is important because it means that even if the optimiser is, in a current implementation, unable to work on a section of code, the result is still a correct program. That is, the optimiser can be constructed incrementally to work on an expanded variety of code and still produce valid results along the way. This amenability to incremental implementation is pleasing, from our perspective as implementors.

Once the wish-list is in-place the optimiser rolls upwards and outwards. The current wish-list becomes the new focus of optimisation and the code immediately upstream is modified to assume the role of the new wish-list. This effect can be seen in part (b) of figure 45 by observing how the underlined code moves and and also by looking at the optimised code to its left.

Note that line (2) of part (b) contains a wish-list that is fragmented over two sections of code inside an alltup function. With the exception of function composition, second-order functions, including alltup, allvec, if, reduce, and while, that contain multiple functional components, open a new front of optimisation for each of their functional components. Each functional component is optimised independently and the wish-lists that result are coalesced to form a new, aggregated, wish-list immediately upstream of the second-order function.

The process of modifying code and creating wish-lists finishes when the most upstream part of the program is reached. Any wish-list that remains at the end of this process is left in place to become part of the program code.

To this point we have described the basic optimisation strategy, based on the propagation of a wish-list upstream, common to both tuple and vector optimisation.

Next, we examine the smaller-scale techniques, the tactics, used by both types of optimisation, that keep the optimisation process tractable.

5.3 Common tactics

Though the basic strategy of optimisation is simple, the details of applying it are not. The problem lies in the unbounded number of equivalent forms that any function can take[4]. The optimiser works best if at each stage of optimisation, every possible case is captured and profitably handled by some rule. However, unless steps are taken to reduce the number of cases, defining an effective optimiser becomes infeasible.

Two, slightly-overlapping, strategies are employed to reduce the number of cases:

[4]Any function can be expressed in equivalent form by composing it with an id function. This transformation can be carried out an unbounded number of times without affecting the semantics of the function. Moreover, there are an unbounded number of ways to express an identity function using combinations of alltup, id and π functions.

- It is made easy for rules to ignore code that is not currently interesting, thus narrowing the window of code each rule has to match.
- Code is kept in a predictable form.

We focus on these two strategies in turn.

5.3.1 Making it easy to ignore uninteresting code

At any intermediate stage of optimisation a BMF program has the following, idealised, form.

$$< \textit{optimised-part} > \cdot < \textit{unoptimised-part} >$$

This tells us that a program consists of an optimised part composed with an unoptimised part. Only the unoptimised part, which, for convenience, includes the wish-list, is of current interest to the optimiser. We can make things even better by noticing that only the downstream part of the unoptimised part (which also includes the wish-list) is, usually, of interest at any one time, so we have:

$$\begin{array}{l} < \textit{optimised-part} > \cdot \\ < \textit{interesting-unoptimised-part} > \cdot \\ < \textit{uninteresting-unoptimised-part} > \end{array}$$

Having a small focus of interest is a good thing. All other things being equal, it is easier to write rules for smaller sections of code than for large sections of code. Another way of saying this is, fewer rules are needed to comprehensively capture the cases presented by small sections of code than larger sections of code[5].

The questions remain as to how code can be conveniently partitioned into interesting and uninteresting parts and how such partitioning can be maintained. These two questions are addressed next.

[5]A basic argument for this assertion can be made numerically. In a language with n different primitive functions a function f, containing no-second order terms, can be adequately handled by n rules. In contrast a function $f \cdot g$ could require up to n^2 rules and $f \cdot g \cdot h$ could require up to n^3 rules and so on. Of course, natural constraints in how functions can be combined will reduce this variety. It must also be noted that this assertion has much less strength if the code has been produced and/or pre-processed in such a way as to dramatically decrease the variety of forms. More will be said about this shortly.

Partitioning with function composition Consider the following, abstractly represented, BMF program:

$$O_0 \cdot O_1 \cdot O_2 \cdot O_3 \cdot I_0 \cdot I_1 \cdot I_2 \cdot U_0 \cdot U_1 \cdot U_2 \cdot U_3$$

The letters represent BMF functions composed together to form a program. The functions O_0 to O_3 are the part of the program that is already optimised. I_0 to I_2 are the unoptimised functions which are of current interest to the optimiser. U_0 to U_3 are the unoptimised functions which are not yet of interest to the optimiser. A human reading the program above can, and should, see a sequence of composed functions with no particular associativity. Function composition is an associative function so where we draw the parentheses, if any, has no effect on program semantics. The Adl implementation represents function composition as a binary operator and because it is an operator of fixed arity, it must choose, at least in its internal representation, where to draw the parentheses. Like the human reader, the Adl implementation has some freedom in doing this. For example, for the program above the Adl implementation could write:

$$((((((((((O_0 \cdot O_1) \cdot O_2) \cdot O_3) \cdot I_0) \cdot I_1) \cdot I_2) \cdot U_0) \cdot U_1) \cdot U_2) \cdot U_3)$$

or it could write:

$$(O_0 \cdot (O_1 \cdot (O_2 \cdot (O_3 \cdot (I_0 \cdot (I_1 \cdot (I_2 \cdot (U_0 \cdot (U_1 \cdot (U_2 \cdot U_3))))))))))$$

or with any other pattern of matching parentheses without affecting program semantics. However, though they are semantically neutral, the brackets can still be used to capture useful information. Brackets can help *guide* rules to the interesting code. For example, if parentheses are drawn the following way:

$$(((O_0 \cdot O_1) \cdot O_2) \cdot O_3) \cdot (((I_0 \cdot I_1) \cdot I_2) \cdot (((U_0 \cdot U_1) \cdot U_2) \cdot U_3))$$

we can easily write a rule to focus on the interesting part:

$$\frac{\mathit{Process\_interesting\_part}(\mathit{bexp}_I \Rightarrow \mathit{bexp}_R)}{\mathit{bexp}_O \cdot (\mathit{bexp}_I \cdot \mathit{bexp}_U) \Rightarrow \mathit{bexp}_O \cdot (\mathit{bexp}_R \cdot \mathit{bexp}_U)}$$

If the rule above were applied to the previous program:

- bexp_O matches $(((O_0 \cdot O_1) \cdot O_2) \cdot O_3)$

- $bexp_I$ matches $((I_0 \cdot I_1) \cdot I_2)$
- $bexp_U$ matches $(((U_0 \cdot U_1) \cdot U_2) \cdot U_3)$

which is precisely what we want. The association we have chosen for composition helps the rule to focus on the I terms. Note that the rule is unaffected by how parentheses are drawn within each group of O, I and U terms. As long as the outer parentheses are drawn correctly the rule will work. Of course, ensuring that the outer parentheses are drawn in the required order requires work and this, the second, issue is addressed next.

Keeping brackets in their place Thus far, we have shown how we can use, otherwise unimportant information, the association of function composition, to partition code into interesting and uninteresting sections. The key to preserving this partitioning information is for each rule, that deals with more than one partition, to both recognise the partitions in its antecedent and produce partitions in its result. In the Adl implementation the most common example of partitioning is between code to the left of the wish-list, the wish-list itself and code to the right of the wish-list. All rules that work with code that straddles the boundary of a wish-list are responsible for producing a new wish-list, and its required context, in their conclusion.

Both stages of the optimiser make extensive use of function composition to partition code. Next, we examine how the optimiser tries to keep code predictable.

5.3.2 Keeping code predictable

It is important to keep code predictable. For any part of the optimiser, the greater the variety of code it encounters, the more sophisticated that part of the optimiser needs to be. This relationship holds in general, varied environments require highly sophisticated, or adaptable, systems to deal with them[2][6]. The cost of making our system sophisticated enough to deal with arbitrary BMF code is too great and making it adaptable is beyond the scope of this research. As a practical option we chose to

[6]We should, perhaps, add that systems that have low levels of sophistication but are highly insensitive to change can also cope with a varied environment. However, by the metrics of the cited article, a system that is simple but is adapted to an environment is still more effectively complex that a system that is sophisticated but is not adapted to the environment. That is, it makes sense to measure the complexity of a system with reference to how much of that complexity helps it in the current environment.

decrease the variability in our system's environment. Since the environment for the optimiser is BMF input code, reducing variability means keeping or making that BMF code more predictable.

This section describes general techniques used by both the vector and whole-value optimisers to ensure that the code handled by our rules is predictable. The main techniques used are:

- Using un-altered translator code.
- Standardising the associativity of code.
- Code compaction.
- Removing surplus identity functions.

The first of these is simply taking advantage of a feature that it already there (unaltered translator code). The last three are forms of normalisation. These techniques are described in turn.

Using unaltered translator code The Adl to BMF translator constructs BMF code using standard templates. The very first stage of the optimiser can be constructed from a small number of rules that match these templates. These templates are quite large and they allow rules to make assumptions that they otherwise wouldn't.

Unfortunately, the structure of translator code quickly disappears during optimisation, which puts strict limits on the extent to which this structure can be exploited. In more concrete terms, only one phase of optimisation, either tuple, or vector optimisation, can actually exploit the known structure of translator code, because each phase completely obliterates this structure. When deciding whether tuple or vector optimisation should take place first we examined which would most benefit from using translator code. We chose to let vector optimisation run first because it makes heavy use of information about the origins of indexing values. When analysing translator-code, this information about origins can be discerned by simple inspection of addressing functions. However, both optimisation phases obliterate this simple relationship between origins of data and the format of the functions that address it. As a consequence, running tuple optimisation first, would mean that origins of data would have to be established, either, using data-flow analysis, or after

$$\frac{\begin{array}{c} f_2 \cdot f_3 \Rightarrow f_3' \\ f_1 \cdot f_3' \Rightarrow f_3'' \end{array}}{(f_1 \cdot f_2) \cdot f_3 \Rightarrow f_3''} \quad (18)$$

$$\frac{f_1 \cdot f_2 \Rightarrow f_2'}{f_1 \cdot (f_2 \cdot f_3) \Rightarrow f_2' \cdot f_3} \quad (19)$$

$$\vdots$$

Figure 46. Two rules from a rule-set that assumes its input is left-associated and ensures that its output is right associated.

heavy use of normalisation to force code to re-establish a the relationship between addressing functions and data-origins. We do not encounter this problem if tuple optimisation runs last because, unlike vector optimisation, tuple optimisation does not have a need to track, remote origins of data and can thus be easily run after vector optimisation has modified the code.

Next, we examine the other techniques we use to keep code predictable. All of these techniques require the expenditure of effort on the part of the optimiser and are thus forms of normalisation.

Standardising the associativity of composition We saw, in the last section, how the ability to define the associativity of function composition helped the cause of further optimisation by allowing code to be partitioned without changing its semantics. Rule sets can, and do, help maintain and exploit such partitioning. For example, a rule-set may assume that its input code is associated to the left and then systematically associate its output code to the right. This approach makes it trivial for rules to distinguish between code that has been processed and code that has not.

As an example figure 46 shows two rules from an actual optimiser rule set that assumes left-associativity in its input and systematically creates right-associativity in its output. If the rule-set is given a left-associated composed sequence of functions such as:

$$(((a \cdot b) \cdot c) \cdot d) \cdot e$$

then rule 18 will match first with the bindings $\{f_1 \mapsto ((a \cdot b) \cdot c), f_2 \mapsto d, f_3 \mapsto e\}$. The top premise of the rule will process $d \cdot e$ to produce an output function e' that might or might not be a composed function. The second premise makes a call to process:

$$((a \cdot b) \cdot c) \cdot e'$$

which matches rule 18 again and is processed to produce:

$$(a \cdot b) \cdot e''$$

which matches the same rule again and is processed to produce:

$$a \cdot e'''$$

This code can no longer match rule 18 and, if e''' consists of two or more composed functions, it will match rule 19[7]. Rule 19 terminates the processing by pushing a into e''' to produce e''''. This pushing is made more efficient if our rules ensure e''' is associated to the right[8].

Note that the rules in figure 46, require input code that is associated to the left in order process that code completely. The Adl implementation uses a *Left_associate* rule-set that, given input code of arbitrary associativity, produces output code that is uniformly left-associated. *Left_associate* is applied to code prior to calling many rule-sets.

It should also be noted that such pre-processing is not always invoked. Rule-sets exist that do not rely on pre-processing but, in our experience, such rule-sets require more discipline in their construction and are more fragile over time.

Code-compaction The archetypal transformation in the optimiser takes the form:

$$f_1 \cdot f_2 \Rightarrow f_3$$

where f_3 is more efficient[9] than f_1 and f_2 in combination. Unfortunately some important transformations take the more complicated form:

$$f_1 \cdot g \cdot f_2 \Rightarrow f_3$$

[7]If e''' is not a composed sequence of functions then at least one additional rule is needed to capture this case.

[8]Which, in the absence of confounding rules, is ensured by the structure of rule 18.

[9]For input data of interest.

$$\mathsf{K} \cdot f \Rightarrow \mathsf{K} \tag{20}$$

$$\frac{\begin{array}{c} f_1 \cdot g \Rightarrow f_1' \\ \vdots \\ f_n \cdot g \Rightarrow f_n' \end{array}}{(f_1, \ldots, f_n)^\circ \cdot g \Rightarrow (f_1', \ldots, f_n')^\circ} \tag{21}$$

$$\begin{array}{c} \dfrac{f_1 \Rightarrow f_1'}{\pi_1 \cdot (f_1, \ldots, f_n)^\circ \Rightarrow f_1'} \\ \vdots \\ \dfrac{f_n \Rightarrow f_n'}{\pi_n \cdot (f_1, \ldots, f_n)^\circ \Rightarrow f_n'} \end{array} \tag{22}$$

$$\frac{f \cdot g \Rightarrow h}{f * \cdot g* \Rightarrow h*} \tag{23}$$

Figure 47. Key rules for compaction

where the g is intervening code that is important to the operation of f_1[10]. Code compaction is used to normalise g to make it more amenable to necessary analysis and transformation[11].

Code compaction minimises the number of function compositions in the critical path through a section of code. The key rules used for code compaction are shown in figure 47. Where rule 20 applies when a constant function K is composed with a function upstream. The rule eliminates this function, reflecting the fact that the result of K is unaffected the result of any preceding code[12]. Rule 21 compacts an upstream function into an alltup function. The rules listed under number 22 project a component function out of an upstream alltup function[13]. Finally, rule 23 lifts the

[10]A concrete example of such a transformation is: $! * \cdot ((\pi_1, + \cdot (\pi_2, 1)^\circ)^\circ) * \cdot \mathsf{distl} \Rightarrow \mathsf{select} \cdot (\pi_1, (+ \cdot (\mathsf{id}, 1)^\circ) * \cdot \pi_2)^\circ$ where the function the bindings for f_1,g and f_2 are:

$$\{f_1 \mapsto !*, g \mapsto ((\pi_1, + \cdot (\pi_2, 1)^\circ)^\circ)*, f_2 \mapsto \mathsf{distl}\}$$

The function g is vital to the semantics of this program and it has to be properly handled during subsequent transformation.

[11]The code for g in the previous footnote does not require compaction though, quite often, code in this position does.

[12]Applying this transformation can cause a program that previously did not terminate to terminate. This is a property that the user of Adl compiler must be aware of.

[13]The real Adl implementation uses a few rules to systematically decompose alltup functions rather than having a, potentially, infinite number of rules defining pointwise transformations as above.

compaction process to composed map functions.

Apart from reducing the length of composition sequences, which has a quantitative rather than qualitative effect on subsequent optimisation, there are a number of other advantages to code compaction. Given code of the form:

$$f_1 \cdot g \cdot f_2$$

these advantages of code compaction, listed in increasing order of importance, are:

1. Compaction reduces the variety of functions found in g.

2. Compaction makes all address references in g relative to the output data of f_2 rather than some intermediate tuple generated inside g.

3. Compaction ensures that every function in f_1 has an un-shared path of composed functions to the values generated by f_2.

Advantage 1 refers to the fact that compaction completely removes composed sequences of alltup functions from g. This can reduce the number of rules needed in subsequently applied rule-sets. Advantage 2 is a corollary of advantage 1, there can be at most one layer of alltup functions between f_1 and the data produced by f_2 making it much easier to utilise the address references left in the compacted version of g.

Finally, advantage 3 opens the way for unfettered optimisation by producing upstream code that can be transformed without an adverse effect on other downstream functions. In more concrete terms, compaction changes code of the form:

$$(f', f'')^\circ \cdot g$$

where any optimisation of g for f' may affect f'', to:

$$(f' \cdot g, f'' \cdot g)^\circ$$

where optimisation of each component of the alltup can proceed independently. This last advantage greatly simplifies the optimisation process.

Removing redundant identity functions An identity function is redundant if its removal from a program does not affect the semantics of that program. Both translation and some parts of optimisation produce redundant identity functions, usually as sentinel values. These redundant identity functions take up space and, more importantly, make code more difficult for rules to recognise.

Redundant identity functions can appear in many places and can take many forms. In the code:

$$(! \cdot \underline{(\pi_1, \pi_2)^\circ}) * \cdot \mathsf{distl} \cdot (\mathsf{id}, \mathsf{iota} \cdot \# \cdot \underline{\mathsf{id}})^\circ \cdot \underline{\mathsf{id}}$$

the underlined functions are redundant identity functions. Their removal greatly simplifies the code:

$$(!) * \cdot (\mathsf{id}, \mathsf{iota} \cdot \#)^\circ$$

Note that not all id functions are surplus to our needs. The remaining id function in the program above is important to the semantics of the program.

Recognising redundant identity functions The large variety of forms that an identity function can take confounds efforts to remove all redundant identity functions from programs. Identity functions on tuples such as:

$$(\pi_1, \pi_2)^\circ$$

are relatively easily detected. However identity functions can be arbitrarily complex. The code:

$$(\pi_2, \pi_1)^\circ \cdot (\pi_2, \pi_1)^\circ$$

is also an identity function[14], as is:

$$(! \cdot (\pi_1, \pi_2)^\circ) * \cdot \mathsf{distl} \cdot (\mathsf{id}, \mathsf{iota} \cdot \#)$$

Redundant identities of this last form are difficult to eliminate in practice[15]. Fortunately, it is possible to detect, and remove, a large proportion of the redundant identity functions that do arise using a relatively simple set of rules.

The costs and benefits of normalisation We have now outlined the three main forms of normalisation used in the Adl implementation, left-association, code-compaction and the removal of identity functions. All three of these operations

[14]Code compaction converts this form to previous form and thus aids in the detection of identity functions.

[15]Part of the reason for this is that the constituent parts of this code are rarely seen as close together as they are in this example. The various elements are often spread over a larger expanse of code. Code compaction can help make such code easier to recognise and eliminate.

exact a cost in terms of the execution time of the optimiser and only the removal of identity functions directly and consistently contributes to faster target code. The key benefit of these normalisation steps is the way they constrain the code so that it can be adequately handled by a tractable number of optimisation rules. In essence using normalisation is a trade-off between the, significant, compile-time cost of the normalisation itself and the run-time benefit of an effective optimisation process.

This concludes our discussion of the common elements of optimisation. The next two sections describe the two phases that employ these elements: vector optimisation and tuple optimisation.

5.4 Vector optimisation

This section describes the Adl project's vector optimiser for BMF programs. First, we outline the basic strategy employed by the vector optimiser. After that, we describe the details of the vector optimisation process. We postpone any discussion of the effectiveness of vector optimisation until the end of the chapter when we assess the optimisation process as a whole.

5.4.1 Strategy for vector optimisation

We showed, in chapter 3, that BMF code is amenable to incremental transformation using rewrite rules. Both vector optimiser, and the tuple optimiser for that matter, rely heavily on incremental transformation. This vector optimisation strategy is:

> Use incremental code transformations to move data-narrowing functions *on vectors* upstream.

This strategy is a specialisation of the, earlier stated, shared strategy of minimising the data produced upstream so that it more precisely matches the needs of code downstream. We can see this relationship by considering what a data-narrowing operation does. A data narrowing operation converts large data items (vectors) into smaller data items (scalars and vector-elements). If a data narrowing operation can be moved upstream then the code that it leaves in its wake must be changed to compensate for the reduced flow of data. This reduction in data flow typically results in less work being done by the program. This idea is illustrated, in an abstract

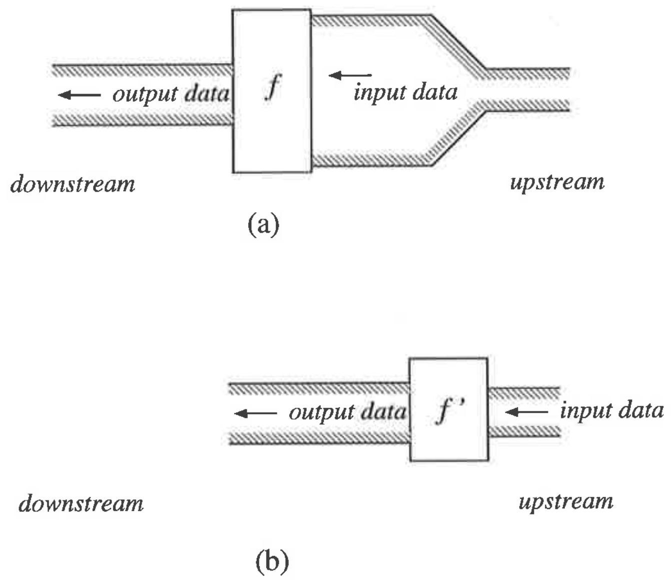


Figure 48. The strategy of moving a data narrowing operation f upstream to reduce the total amount of data movement in the program. Part (a) represents data flow prior to optimisation and part (b) represents data flow after optimisation. Note that f and f' are not necessarily the same function but their output must always be the same.

manner, in figure 48. The figure represents the program as a channel, through which data flows from right-to-left, just as it does in BMF programs[16]. The channel is a series of composed functions. The width of the channel represents the amount of data flowing through the functions in that part of the program. Part (a) represents a program prior to vector optimisation. The program starts with a small amount of data but soon the amount of data increases before encountering a function f. The function f is a data-narrowing operation (on vectors) and its output is very much smaller than its input. This pattern of data-flow is typical of translator code and it is, almost always, non-optimal.

Part (b) represents the data flow through the optimised program. The function f has been moved upstream. In the process, f is transformed into a different function f' that takes in a much smaller quantity of data. In most cases, there is little superficial resemblance between f and f' but they are related by the fact that they must produce the same output given the same initial input to the *program*.

[16]with the exception of while loops which allow data to circulate.

The data narrowing operations There are two data-narrowing operations on vectors:

- Vector indexing (!)
- and vector length (#)

When the vector optimiser encounters these, it attempts to move them upstream. It turns out that moving the # function upstream, as far as the point of creation of its input vector, is usually easy. On the other hand, the index function ! is reliant on both its input vector and its index value and, as such, its movement is beholden to the origins of both of these parameters. As a result, ! is usually much more difficult to move upstream. Consequently, ! is the primary focus of the vector optimiser and the discussion to follow.

At this point, we know the data-narrowing operations and we are aware of the vector-optimisation strategy in abstract terms. Next, we add more depth to this understanding by looking at two, more concrete, examples of the effect of vector optimisation on code.

First example - minor improvement The vector optimiser converts the following unoptimised code

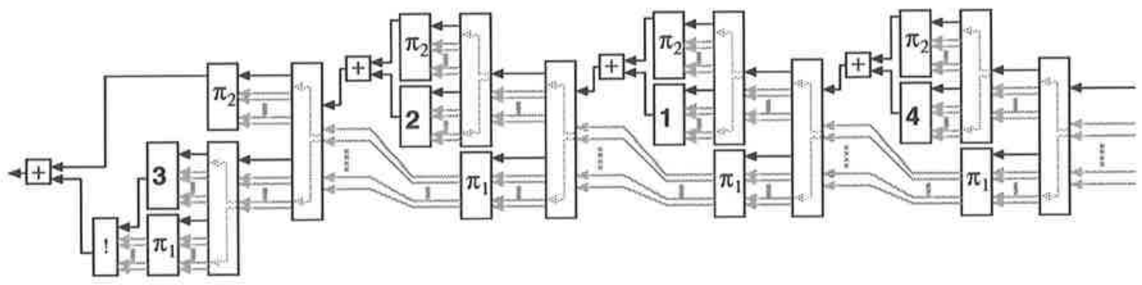
$$+ \cdot (! \cdot (\pi_1, 3)^\circ, \pi_2)^\circ \cdot (\pi_1, + \cdot (2, \pi_2)^\circ)^\circ \cdot (\pi_1, + \cdot (1, \pi_2)^\circ)^\circ \cdot (\pi_1, + \cdot (4, \pi_2)^\circ)^\circ$$

to the more efficient:

$$+ \cdot (! \cdot (\pi_1, 3), + \cdot (2, + \cdot (1, + \cdot (4, \pi_2)^\circ)^\circ)^\circ)^\circ$$

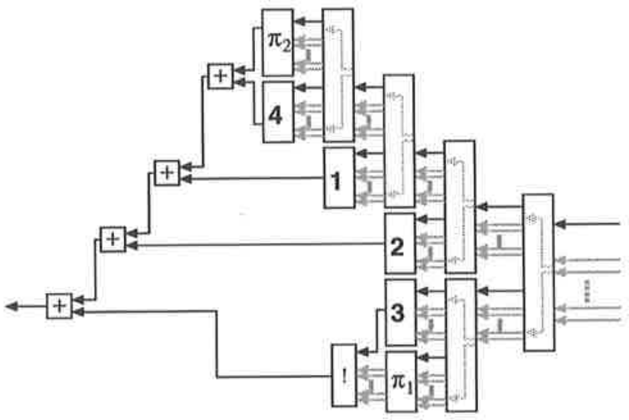
In this case, vector optimisation has the effect of compacting the code. This compaction has the desired effect of bringing the ! function upstream but, as figure 49 illustrates, the gain in efficiency is slight. Each arc in the figure represents the transmission of a data value. The input to the program is a tuple, consisting of a scalar value (black line) and a vector of scalars (grey lines). In this case, the compacted, vector optimised code, in part (b), causes the, potentially large, input vector to ! to flow through fewer functions than it did in part (a). The transformation is of material benefit[17] and there is little more we could expect a vector optimiser to do.

[17]There is still a lot of unnecessary transport of data in the upper portion of part (b)'s diagram. Getting rid of much of this remaining redundant transport is the task of the tuple optimiser, whose description starts on page 143



$+ \cdot (! \cdot (\pi_1, \mathbf{3})^\circ, \pi_2)^\circ \cdot (\pi_1, + \cdot (\mathbf{2}, \pi_2)^\circ)^\circ \cdot (\pi_1, + \cdot (\mathbf{1}, \pi_2)^\circ)^\circ \cdot (\pi_1, + \cdot (\mathbf{4}, \pi_2)^\circ)^\circ$

(a)



$+ \cdot (! \cdot (\pi_1, \mathbf{3})^\circ, + \cdot (\mathbf{2}, + \cdot (\mathbf{1}, + \cdot (\mathbf{4}, \pi_2)^\circ)^\circ)^\circ)^\circ$

(b)

Figure 49. The effect of moving ! upstream in a simple program. The code prior to moving ! is shown in part (a). Part (b) shows the code after moving !.

However, the gain in efficiency is relatively insubstantial compared to the effect of vector optimisation on the next example.

Second example - major improvement Vector optimisation has a much greater effect when there is more redundant data transport defined by the translator code. A typical example where there is a lot of surplus transport is the program

$$(! \cdot (\pi_1, \pi_2)^\circ) * \cdot \mathsf{distl} \cdot (\pi_1, \pi_2)^\circ$$

which permutes the first input vector according to the indices of a second input vector. Vector optimisation converts this to the code:

$$\mathsf{select} \cdot (\pi_1, \pi_2)^\circ$$

The difference this transformation makes can be discerned by comparing the pictures shown in figure 50. Part (a) shows the data movement implicit in the first version of the program. The light-grey lines, are the elements of the source vector and the black lines are elements of the vector of indices. The distl function produces a copy of the source vector for every element of the index vector. At the end of the program only *one* element of each copy of the source vector is used in the result. All of the rest of the elements are replicated needlessly.

Part (b) of figure 50 shows a much more efficient version of the program. The costly combination of replication plus indexing has been eliminated and replaced with the efficient select function.

What is select? select is a function of two vectors. The output of select is a vector containing the elements of the first vector, the source vector, indexed by the second vector, the index vector. An example of how select works is:

$$\mathsf{select}\ [1,2,3,4]\ [3,1,1,2,0] = [4,2,2,3,1]$$

select cannot be efficiently implemented in terms of the other BMF functions presented thus far so, in the Adl implementation, select is a primitive function. select executes in time proportional to the length of the index vector.

Where vector optimisation has most effect The two preceding examples motivate and illustrate the basic strategy of vector optimisation. The effect of vector optimisation is greatest when it applies to code that distributes copies of whole vectors for later indexing. The great majority of such cases are encountered in, and around map functions. The optimisation of map forms the largest part of the vector optimiser.

A note about some examples to come In the Adl implementation, vector optimisation is very strongly focused on optimising access to vector elements. The vector optimiser, for the most part, ignores superfluous references to elements of tuples, on the grounds that eliminating these is the job of tuple-optimisation. As a result of this selective focus, code produced by the vector optimiser can be cluttered and hard-to-read[18]. Such clutter adds nothing to the Adl examples so we omit it,

[18]It can also, in some cases, still be inefficient. This is because, although the explicit references to vector elements have been optimised, the vector as a whole may still be distributed by the code.

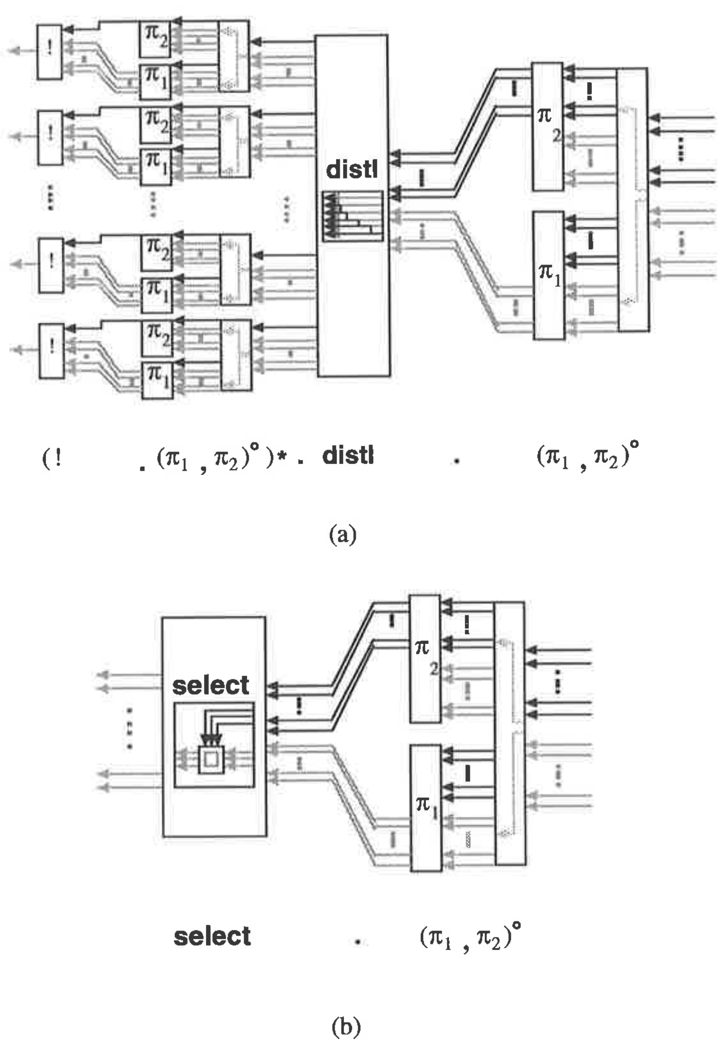


Figure 50. The effect of moving ! upstream into a distl function, causing both functions to be eliminated and replaced with select

in some examples to come, by showing the code, much as it would be after running both vector and tuple optimisation. In no case is this a misrepresentation as, in all examples in this section, the gains in efficiency are almost entirely due to vector optimisation. However, the reader should be aware that the examples omit some messy details.

Summary and preview Thus far we have described the strategy of vector optimisation and presented two examples to illustrate where vector optimisation has most impact. The time has come to describe the operation of the vector optimiser in detail. The next section describes the most challenging part of vector optimisation, the optimisation of map. After that we will devote a, much shorter, section to describing vector optimisation of all other functions.

5.4.2 Vector Optimisation of Map

The strategy for optimising map mirrors the vector optimisation strategy shown in figure 48. The translator code for map behaves in a way that corresponds to part (a) where the distl function, upstream of the map, is responsible for widening the data and ! or # functions inside the map are responsible for narrowing it. Optimisation of map brings these functions together and eliminates them to produce a program with behaviour corresponding to part (b). Figure 50 gives the archetypal version of the optimisation process for map. Part (a) shows an inefficient program with both distl and ! functions present. Part (b) eliminates them both, leaving a select function in their place. This transformation strategy is simple and, in this case, so is the rule to implement it.

Given this simple strategy it might seem that the process for vector optimising map is straightforward. Unfortunately, rules such as the one in figure 50 are, typically, only applicable after long sequence of transformations to pre-process code to a form that matches the rule. In general, a map function can contain arbitrarily many references to one or more elements of one or more vectors. Moreover, map functions over vectors can be nested, resulting in interactions between levels of indexing that defy simple characterisation. To cope with this complexity, the rules for performing vector optimisation on map are the most detailed in the optimiser.

This distribution is often left in place because the tuple optimiser has not had the chance to establish that the vector, as a whole, does not need distribution.

In the explanations ahead we avoid a strong emphasis on the low-level details of how map functions are optimised. Instead, the focus will be on the core components of the optimisation process and on how translator code is processed so these components can be widely applied. Where possible, we illustrate the effect of this process with examples.

The structure of this section is as follows. Next, we describe the anatomy of the translator code produced from `map` functions in Adl and define the terminology for the rest of this section. Second, in section 5.4.2.2 we outline the phases of vector optimisation of map functions. Section 5.4.2.3 describes the core of the optimisation process for indexing functions. Indexing functions are the key focus of vector optimisation so more space is devoted to this section than others. Finally, section 5.4.2.4 outlines how other functions, besides indexing functions, appearing in map functions are vector optimised.

5.4.2.1 Anatomy of Translated Calls to Map Functions

The translator, described in the previous chapter, converts Adl calls of the form:

$$\texttt{map}(f, e)$$

to:

$$(B) * \cdot \mathsf{distl} \cdot (\mathsf{id}, R)^{\circ}$$

Where B is the translation of the right-hand-side of the function f and R is the translation of the expression e which is responsible for generating the input vector. The id function captures an environment containing all the global values in the static scope of f. The distl function distributes this environment across the input vector[19] and the $()*$ function, encapsulating B, ensures that B is applied to every element of the input vector. The whole of this BMF code is called the *map-translation* and we

[19]Distribution of a similar nature occurs in the FARM skeleton[42] which is defined:

$$\begin{array}{l} FARM :: (\alpha \to \beta \to \gamma) \to \alpha \to [\beta] \to [\gamma] \\ FARM\ f\ env\ xs = map\ (f\ env)\ xs \end{array}$$

where the environment *env* is partially applied to f and implicitly distributed across the array xs. One difference between the *FARM* skeleton and the *map-translation* is that the programmer has some control over what is contained in *env* in the *FARM* skeleton whereas Adl source code provides no such control. Another difference is that *env* is destined to be the same for all instances of the *map* function inside the *FARM* skeleton but, in the Adl translation, the explicitly distributed environment can be, and is, tailored for each invocation of B by the vector optimiser.

will refer to it by this name in the remainder of this report. We will also use the terminology *map-body* to refer to B and the term *range-generator* to refer to R.

For the purposes of this discussion the most significant feature of the map-translation is that the map-body is given all of the values in scope just in case they are needed. This is inefficient and the task of the optimiser is to reduce the size of this set of values. The task of the vector optimiser is more specific - The vector optimiser tries to reduce quantity of vector elements flowing into the map body by moving data-narrowing functions such as vector-indexing and vector-length out of the map-body. We describe how the vector optimiser approaches this task next.

5.4.2.2 Phases of Map Optimisation

Vector optimisation of map is carried out in three phases:

1. **Pre-Processing:** normalises the functions in map-translation to allow them to be captured by the core-processing rules that follow.
2. **Core-processing:** applies transformations necessary to move indexing and length functions outside of the map-body.
3. **Post-Processing:** consolidates the products of core-processing to produce a cleanly structured map function and a single wish-list for further processing.

We briefly describe each of these phases in turn.

Preprocessing Before the main work of optimisation can take place and before we can apply the rules for the cases we will consider shortly, the map-translation must be pre-processed. Pre-processing assumes that all ! and # functions which are amenable to further processing are already in the wish-list inside the map body. However, these functions may be buried at some depth in the wish-list, which may obscure the values that they are referencing. Pre-processing uses code-compaction to compress the code immediately upstream to make ! and # functions more easily accessible and resolve the values they are referencing. Figure 51 gives a concrete example of this process. Part (a) of the figure shows the map-body, with wish-list underlined, prior to pre-processing. The values being referenced by the # and ! functions are not readily apparent to the optimiser because of the sequence of composed alltups immediately upstream of these functions. Pre-processing compacts this upstream code to produce

$$(f \cdot \underbrace{(\# \cdot \pi_1 \cdot (\pi_2, \mathsf{id})^\circ, ! \cdot (\pi_1, \pi_1 \cdot \pi_2)^\circ \cdot (\pi_2 \cdot \mathsf{id})^\circ)^\circ \cdot (\pi_2, \pi_1)^\circ}_{(a)})*$$

$$(f \cdot \underbrace{(\# \cdot \pi_1 \cdot \pi_1, ! \cdot (\pi_1 \cdot \pi_1, \pi_2)^\circ)^\circ}_{(b)})*$$

Figure 51. The effect of pre-processing on a map body. Part (a) shows a map body before pre-processing. The wish-list (underlined) is quite long, the # and ! functions are nested several compositions deep making it impractical to define a simple rule to access them. Part (b) shows the same code after pre-processing. The # and ! functions are nested less deeply and the positions of the values they reference in the input tuple are easily resolved. Note that f is a section of previously optimised code.

the code in part (b). The compaction process has removed intermediate layers of functions that previously obscured the values being referenced in the functions producing the parameters to ! and #. After compaction these functions, and their immediate parameters can be accessed, and processed, easily and independently by the core-processing stage.

Note that, for indexing functions, compaction is not the end of the pre-processing phase. In certain cases the input vector is re-oriented, and the indexing functions referencing it are adjusted, to make code more amenable to core processing The details of this reorientation process will be described later, starting on page 127.

Core-processing Core-processing attempts, by a process of incremental transformation, to pull index and length functions, now residing in the most upstream part of the map body, further upstream - out of the map-body entirely. The benefit of moving these data-narrowing functions out of the map body is that their, potentially large, input data does not have to be distributed to each instance of the map body. This, typically, leads to much reduced data-replication costs during program execution.

In almost all cases, pulling a length function out of the map-body is straightforward[20] and we will not discuss this process further. In contrast pulling an index function out of map-body is not always a trivial process. The code resulting

[20]As long as the vector being processed by the length function is not generated inside the map-body. It is trivially replaced by a reference to the result of an application of the length function to the same vector upstream of the map-body.

from this movement also varies in complexity according to the origins of the index values and the presence or absence of nested index functions. Core processing of index-functions is described in section 5.4.2.3. Core-processing of other non-index functions is described in section 5.4.2.4.

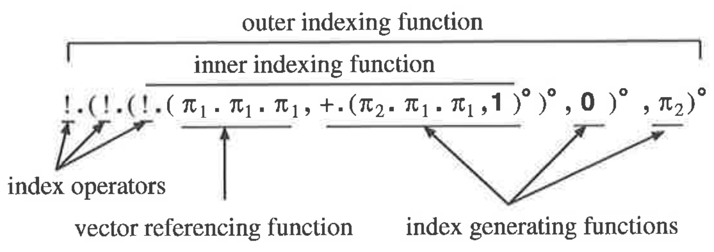
Post-processing Core-processing optimises each index and length function separately producing an alltup of isolated map functions which are consolidated further downstream using one or more zip functions. Post-processing works to re-unite these functions and their upstream code in order to improve readability and slightly improve efficiency. Post-processing also has the effect of producing a single wish-list for use in further optimisation of upstream code. Post-processing contains non-trivial code and plays an important house-keeping role in optimisation. We do not provide a separate description of the post-processing rules for the vector optimisation of map here but note their effect, where appropriate, in our descriptions of the core vector optimisation process.

This concludes our brief overview of the stages of vector optimisation of map-translations. The most novel and challenging aspects of this work relate to the optimisation of indexing functions, and we describe these next.

5.4.2.3 Core Processing of Indexing Functions

The optimisation steps performed in the core processing of indexing functions in map-translations are the ultimate source of most of the benefit derived from vector optimisation. At the centre of this optimisation process lie a few simple index processing rules. These rules, combined with the correct infrastructure to apply them, are very effective in reducing data movement in a broad range of indexing functions. For some nested index functions, the effectiveness of the index processing rules is further improved with judicious reorientation of input vectors prior to the application of these rules. Although, it is actually a form of pre-processing we delay any further discussion of the reorientation process until page 127, after we have described the index-processing rules. We do this because the description of the index processing rules makes the motivation for the reorientation process clearer.

The layout of the remainder of this section is as follows. First, we explain the anatomy of an indexing function, defining terms for its parts. Second, we present the index-processing rules and describe how each is applied and describe their impact on



outer indexing function
inner indexing function
!.(!.(!.(π1.π1.π1,+.(π2.π1.π1,1)°)°,0)°,π2)°
index operators
vector referencing function
index generating functions

Figure 52. An example of a nested indexing function.

the efficiency of the program. Third, we describe the reorientation process to improve the impact of the index-processing rules for some indexing functions. Fourth, we describe how the vector optimiser handles map-translations with multiple indexing functions. Finally, we describe how map-translations containing conditional functions with embedded vector indexing functions that are handled.

Anatomy of Vector Indexing Functions Indexing functions have the form:

$$! \cdot (\textit{vector-referencing-function}, \textit{index-generating-function})^\circ$$

The ! symbol is a binary *index-operator*[21]. It takes a pair consisting of a vector argument and an integer index and returns the element of the vector referenced by the index. The *vector referencing function* generates the vector argument to the index-operator and the *index generating function* generates the index argument to the index operator.

Figure 52 shows an example of a nested indexing function with labels describing its various parts. Note that, in this example, the inner indexing functions act as the vector referencing function for the outer indexing function. This layout is typical of indexing functions applied to nested vectors.

Now that we have defined the nomenclature for indexing functions we can describe the index processing rules.

Index processing rules There are three rules for processing vector indexing functions found in the map-bodies of map-translations. These are described in figure 53. These rules are derived from the *IOpt* rule-set of the optimiser definition[22]

[21]Occasionally, in other parts of this report the !-symbol is referred to as an *index-function* as distinct from an *indexing-function* which is the index-operator combined with the alltup function immediately upstream. To avoid confusion, in this section, the !-symbol is called the index-operator. In other sections the use of *index-function* is further clarified by its context.

Set *IOpt* is $\boxed{b\_exp \Rightarrow b\_exp}$

Select introduction

$$\frac{\begin{array}{c} Not\_Member\_of\_oexp(b\_exp_1, \pi_2) \\ Member\_of\_oexp(b\_exp_2, \pi_2) \\ Opt(b\_exp_2 \Rightarrow b\_exp_2') \end{array}}{\begin{array}{l} (! \cdot (b\_exp_1, b\_exp_2)^\circ) * \cdot \mathsf{distl} \cdot (V, R)^\circ \Rightarrow \\ \qquad \mathsf{select} \cdot (b\_exp_1, b\_exp_2')^\circ \cdot (V, R)^\circ \end{array}} \tag{24}$$

Repeat introduction

$$\frac{\begin{array}{c} Not\_Member\_of\_oexp(b\_exp_1, \pi_2) \\ Not\_Member\_of\_oexp(b\_exp_2, \pi_2) \end{array}}{\begin{array}{r} (! \cdot (b\_exp_1, b\_exp_2)^\circ) * \cdot \mathsf{distl} \cdot (V, R)^\circ \Rightarrow \\ \mathsf{repeat} \cdot (! \cdot (b\_exp_1, b\_exp_2)^\circ, \# \cdot \pi_2)^\circ \cdot (V, R)^\circ \end{array}} \tag{25}$$

Catch-all rule

$$\frac{\begin{array}{c} Member\_of\_oexp(b\_exp_1, \pi_2) \\ Opt((b\_exp_1, b\_exp_2)^\circ \Rightarrow b\_exp) \end{array}}{\begin{array}{l} (! \cdot (b\_exp_1, b\_exp_2)^\circ) * \cdot \mathsf{distl} \cdot (V, R)^\circ \Rightarrow \\ \quad (!) * \cdot b\_exp \cdot (V, R)^\circ \end{array}} \tag{26}$$

End *IOpt*

Figure 53. Index processing rules.

which is a specialised rule-set for handling indexing-functions which resides inside the larger *Opt* rule set used for core processing of all functions in map-translations. Rule 24 applies when there is an immediate opportunity to replace the indexing and distl function with a much-more efficient select function. Rule 25 applies when the indexing function produces a value that is invariant across all invocations of the map body. In this case, the indexing function can be dragged upstream of the map-body and a copy made, using repeat, in place of each invocation of the original indexing function. repeat is defined:

[22]It should be noted that, for clarity of exposition, more context has been added to the rules above than appears in the actual rules in the optimiser. The actual optimiser rules do not include the distl and $(V, R)^\circ$ functions. This makes it easier to apply the rules in situations where the map-body contains more than just an indexing function. The rules of the optimiser re-inject the results of its rules back into context after the it applies the rules from *IOpt*. None of this affects the validity of the rules in figure 53.

$$\begin{aligned}\mathsf{repeat}\,(v,n) &= [v]{+\!\!+}\,\mathsf{repeat}(v,n-1), n>0\\ \mathsf{repeat}\,(v,0) &= [\,]\end{aligned}$$

That is, repeat takes a pair consisting of a value v and an integer i and produces a vector $[v,\ldots,v]$ containing i copies of v. The last rule (rule 26) applies when there are no immediate and easily detectable opportunities for further optimisation.

Now, having described the purpose of each rule, we now explain how and why the premises of each rule determine when it is applied.

Rule choice For a given piece of code, the rule from figure 53 to be applied depends on the the presence or absence of π_2 functions in either the vector-referencing-function b_exp_1 or the index-generating-function b_exp_2. The predicate *Member_of_oexp* is true if its second parameter, in this context: π_2, appears as a most upstream function in its first parameter. The predicate *Not_Member_of_oexp* is true if its second parameter, again π_2, does *not* appear as a most upstream function in its first parameter. The significance of π_2 in these rules lies with the fact that, in the context of a map-body, a π_2 function, appearing as a most upstream function, must always reference the range-generator of the encapsulating map-translation. In these rules the range-generator is denoted R. Any function that references R *can* vary between invocations of the map-body. Conversely, any function that does not reference R *cannot* vary between invocations of the map-body. This variance, or lack thereof, is pivotal in deciding whether it is appropriate to use select (rule 24) or repeat(rule 25) or not to insert either of these two (rule 26).

Scope for further optimisation The premises of the rules in figure 53 call the *Opt* rule-set on every term that *might* have π_2 as a most-upstream function. Conversely, they neglect to call *Opt* on any term known not to have π_2 as a most upstream function. The reason for this pattern of calls is that any function *without* a most-upstream π_2 function can, during post-processing, be moved into the wish-list for further processing at some later stage. Such functions do not need to be optimised immediately. In contrast, functions that contain π_2 as a most upstream function, by virtue of the fact that they access a value created in R, the range-generator, cannot be moved further upstream for processing at a later stage. Such functions must be processed in-situ, hence the call to *Opt* on these functions.

$$(! \cdot (\pi_1, \pi_2)^\circ) * \cdot \mathsf{distl} \cdot (\mathsf{id}, \mathsf{iota} \cdot 4)^\circ$$

Figure 54. A map-translation containing an indexing function amenable to application of rule 24 (select-introduction)

$$\mathsf{select} \cdot (\pi_1, \pi_2) \cdot (\mathsf{id}, \mathsf{iota} \cdot 4)^\circ$$
$$(a)$$

$$\mathsf{select} \cdot (\mathsf{id}, \mathsf{iota} \cdot 4)^\circ$$
$$(b)$$

Figure 55. The program shown in figure 54 immediately after application of rule 24 (select-introduction) (part (a)) and after subsequent cleanup (part (b)).

We have now, briefly described the three rules of *IOpt*. These rules are referenced frequently in the following discussion and so, for the sake of clarity, we will sometimes suffix references to these rules with the corresponding name from figure 53.

Examples applying rule 24: Select Introduction Rule 24 applies whenever the index-generating-function depends on the range-generator and the vector-referencing-function does not. A simple example where this condition is true is shown in figure 54. In this code the index-generating-function, π_2, depends on the range-generator, $\mathsf{iota} \cdot 4$, and the vector-referencing-function, π_1 does not. When rule 24 is applied to this code it is transformed to the code in figure 55(a), which is trivially cleaned-up in subsequent processing to produce the code in part (b). The code in figure 55 avoids distribution of its input vector and is thus significantly more efficient than the code in figure 54.

Figure 56 shows a program containing a nested map-translation containing a nested indexing function[23]. The inner map-translation (underlined) is applicable to rule 24. Figure 57(a) shows the same program after the application of rule 24 to the inner map-translation. Part (b) shows the same program after subsequent cleanup of the inner map-translation using simple BMF identities to merge $(!\cdot(\pi_1\cdot\pi_1, \pi_2\cdot\pi_1)^\circ, \pi_2)^\circ$

[23]Though it is not of vital significance, in this context, the program produces a square-shaped subset of its input vector.

$$\underline{((! \cdot (! \cdot (\pi_1 \cdot \pi_1, \pi_2 \cdot \pi_1)^\circ, \pi_2)^\circ) * \cdot \mathsf{distl} \cdot (\mathsf{id}, \mathsf{iota} \cdot \# \cdot \pi_1)^\circ)} * \cdot \mathsf{distl} \cdot (\mathsf{id}, \mathsf{iota} \cdot \# \cdot \mathsf{id})^\circ$$

Figure 56. A nested map-translation, amenable to rule 24(select-introduction), containing a nested indexing function. The inner map-translation is underlined.

$$(\mathsf{select} \cdot (! \cdot (\pi_1 \cdot \pi_1, \pi_2 \cdot \pi_1)^\circ, \pi_2)^\circ \cdot (\mathsf{id}, \mathsf{iota} \cdot \# \cdot \pi_1)^\circ) * \cdot \mathsf{distl} \cdot (\mathsf{id}, \mathsf{iota} \cdot \# \cdot \mathsf{id})^\circ$$
$$(a)$$

$$(\mathsf{select} \cdot (! \cdot (\pi_1, \pi_2)^\circ, \mathsf{iota} \cdot \# \cdot \pi_1)^\circ) * \cdot \mathsf{distl} \cdot (\mathsf{id}, \mathsf{iota} \cdot \# \cdot \mathsf{id})^\circ$$
$$(b)$$

$$(\mathsf{select} \cdot (\pi_1, \mathsf{iota} \cdot \pi_2)^\circ \cdot \underline{(! \cdot (\pi_1, \pi_2)^\circ, \# \cdot \pi_1)^\circ}) * \cdot \mathsf{distl} \cdot (\mathsf{id}, \mathsf{iota} \cdot \# \cdot \mathsf{id})^\circ$$
$$(c)$$

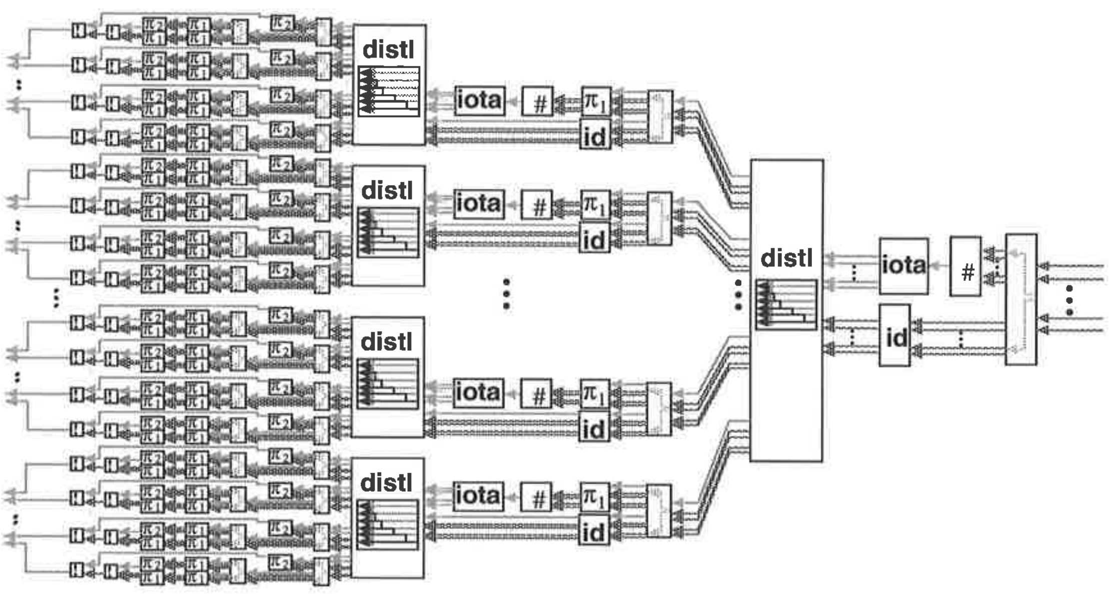
Figure 57. The program in figure 56 after application of rule 24 (select-introduction) to the inner map-translation (part (a)), after subsequent cleanup (part (b)) and after post-processing (part (c)) to extract a new wish-list (underlined).

and $(\mathsf{id}, \mathsf{iota} \cdot \# \cdot \mathsf{id})^\circ$. Part (c) shows the same program with the inner map-translation further post-processed to have a distinct wish-list (underlined). This wish-list includes, as one of its most-upstream functions, an indexing function, $! \cdot (\pi_1, \pi_2)^\circ$, that can be the subject of further optimisation using rule 24. Figure 58 shows the program in figure 57 (c) after the application of rule 24[24] and subsequent post-processing. An intuition of the efficiency of this program relative to the unoptimised version in figure 56 can be had by comparing figure 59 with figure 60. Figure 59 contains a schematic of the the data flows in the unoptimised program of figure 56. Note that, in this figure, to save space, we use the darker lines to denote entire vectors;

[24]Note again, that the actual versions of the rules in *IOpt* remove indexing functions from their immediate context which makes possible the direct application of rule 24 to map-translations like the one in figure 57 where indexing functions are mixed in with other functions.

$$(\mathsf{select} \cdot (\pi_1, \mathsf{iota} \cdot \pi_2)^\circ) * \cdot \mathsf{distl} \cdot (\pi_2, \pi_1)^\circ \cdot (\mathsf{select} \cdot (\mathsf{id}, \mathsf{iota} \cdot \# \cdot \mathsf{id})^\circ, \# \cdot \mathsf{id})^\circ$$

Figure 58. The program in figure 57 with both the inner and outer map-translations optimised.



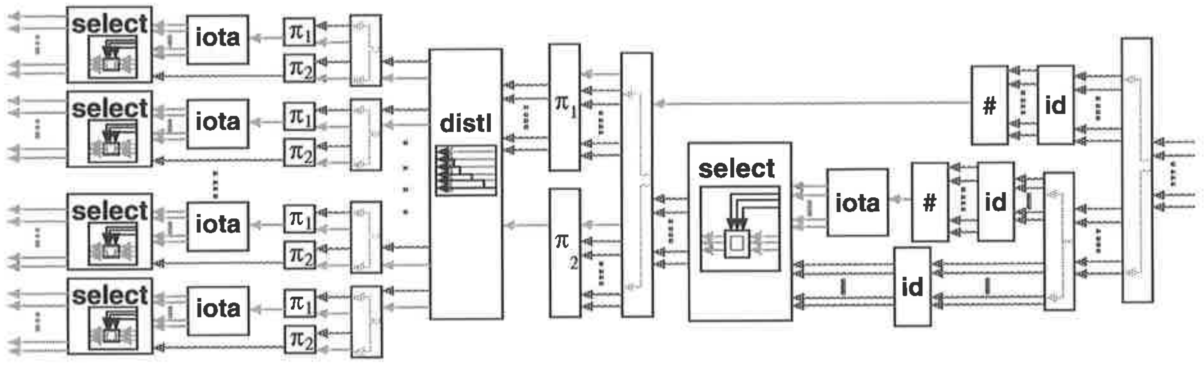
$((\,!\,.\,(!\,.\,(\pi_1\,.\,\pi_1,\pi_2\,.\,\pi_1)^\circ,\pi_2)^\circ\,)*\,.\,\mathbf{distl}\,.\,(\mathbf{id},\mathbf{iota}\,.\,\#\,.\,\pi_1)^\circ\,)*\,.\,\mathbf{distl}\,.\,(\mathbf{id},\mathbf{iota}\,.\,\#\,.\,\mathbf{id})^\circ$

Figure 59. Data flows in the unoptimised nested map-translation shown in figure 56. Note that each darker line in this diagram represents an entire vector.

the lighter lines denote individual values. The input vector is nested. The diagram shows how, even with the small number of actual input values shown, data flow increases dramatically as the program executes. Downstream parts of the program are so crowded that most individual arcs cannot be discerned. It is only at the end, with the application of nested vector indexing operations, that data flow dramatically decreases.

Figure 60 shows the data flows in the optimised program in figure 58. The moderate data flows in this program stand in stark contrast to the vast quantities of data handled by the unoptimised program. The use of select, instead of the distl/indexing combination, greatly reduces the replication of data. Almost all of this reduction is attributable to the application of rule 24. Next we look at an application of rule 25 (repeat-introduction), which also leads to significant gains in efficiency.

Examples applying rule 25: Repeat Introduction Rule 25 applies when neither the index-generating-function nor the vector-referencing-function depend on the range-generator. In these cases the result of the indexing function is constant across all invocations of the map-body and can be replaced by repetition of its result.



$(\mathsf{select}.(\pi_2,\mathsf{iota}.\pi_1)^\circ)*.\mathsf{distl}.\,(\pi_2,\pi_1)^\circ.(\mathsf{select}.(\mathsf{id},\mathsf{iota}.\#.\mathsf{id})^\circ,\#.\,\mathsf{id})^\circ$

Figure 60. Much improved data flows in the optimised map-translation shown in figure 58. Note that each darker line in this diagram represents an entire vector.The programs shown in this figure and figure 59 are semantically equivalent.

$$(! \cdot (\pi_1, 0)^\circ) * \cdot \mathsf{distl} \cdot (\mathsf{id}, \mathsf{id})^\circ$$

Figure 61. A map-translation containing a vector index function with a constant index parameter.

Figure 61 contains a map-translation to which rule 25 applies. In this program the index-generator is a constant function, 0. Rule 25 converts this code into the code shown in figure 62(a). Post-processing further simplifies the code to that shown in part (b). Note that the versions of the program shown in figure 62 are much more efficient than the original map-translation in figure 61 because the latter replicates the entire vector for every invocation of the map-body, using distl, whereas the optimised program replicates the result of the indexing function, using repeat.

Another example where rule 25 is applicable is shown in figure 63. In this code the

$$\mathsf{repeat} \cdot (! \cdot (\pi_1, 0)^\circ, \# \cdot \pi_2)^\circ \cdot (\mathsf{id}, \mathsf{id})^\circ$$
(a)

$$\mathsf{repeat} \cdot (! \cdot (\mathsf{id}, 0)^\circ, \# \cdot \mathsf{id})^\circ$$
(b)

Figure 62. Optimised version of the program shown in figure 61

$$(! \cdot (\pi_1 \cdot \pi_1, \pi_2 \cdot \pi_1)^\circ) * \cdot \mathsf{distl} \cdot (\mathsf{id}, \pi_1)^\circ$$

Figure 63. A map-translation containing an index function with an index parameter that is constant for each invocation of the map-body

$$\mathsf{repeat} \cdot (! \cdot (\pi_1 \cdot \pi_1, \pi_2 \cdot \pi_1), \# \cdot \pi_2)^\circ \cdot (\mathsf{id}, \pi_1)^\circ$$
(a)

$$\mathsf{repeat} \cdot (! \cdot (\pi_1, \pi_2)^\circ, \# \cdot \pi_1)^\circ$$
(b)

Figure 64. Code from figure 63 after the application of rule 25 (repeat-introduction) (part (a)) and after a small amount of post-processing (part (b)).

input data is a pair (v, x) where v is a vector and x is an integer index. The index-generator, $\pi_2 \cdot \pi_1$ in this code references the input data x and, thus, is invariant across all invocations of the map-body. The application of rule 25 to the code in figure 63 produces the code in figure 64(a) and, after a small amount of post-processing, then produces the code in figure 64(b). Again the optimised versions of the program are very much more efficient than the corresponding translator code because they avoid distributing copies of the input vector.

We have now seen map translations applicable to rules 24 (select-introduction) and rule 25 (repeat-introduction). Next we look at the final, Catch-all, rule from *IOpt*.

Examples applying rule 26: The Catch-all Rule The final rule for core-optimisation of indexing-functions in map-translations is rule 26. This rule applies when π_2 can be found as a most-upstream function in the vector-referencing function. The appearance of π_2 in the vector-referencing function is problematic since it is the vector-referencing function that determines the vector to be indexed. If this function contains π_2 as a most-upstream function it means that the *vector being indexed* can change between invocations of the map-body. This situation, of indexing into changing vectors, cannot easily be characterised by a single function such as repeat or select. The approach taken by the catch-all rule is to leave the index function where it is,

$$\underline{((!\cdot\overset{b\_exp_1}{\overline{(!\cdot(\pi_1\cdot\pi_1,\pi_2)^\circ}},\ \overset{b\_exp_2}{\overline{\pi_2}}\ )^\circ)*\cdot\mathsf{distl}\cdot(\overset{V}{\overline{\mathsf{id}}},\overset{R}{\overline{\mathsf{iota}\cdot\#\cdot\pi_1}})^\circ)}*\mathsf{distl}\cdot(\mathsf{id},\mathsf{iota}\cdot\#)^\circ$$

Figure 65. A map-translation that fills each row in its nested result vector with elements from the diagonal of its nested input vector. Rule 26 (catch-all) is applicable to the underlined code. The sections of code corresponding to variables in rule 26 are labelled.

encapsulated by its surrounding map function, and separately optimise the code that produces its parameters. Figure 65 shows a nested map translation that is captured by rule 26.

Given a square-shaped nested vector this program returns a nested vector, of the same shape, with each row filled with the diagonal elements of the the input vector. The code to which rule 26 is applicable is underlined. The labels above the line denote the code corresponding to variables in rule 26. The first premise of rule 26:

$$Member\_of\_oexp(b\_exp_1, \pi_2)$$

tests for the presence of a π_2 function in the most-upstream parts of b_exp_1: $!\cdot(\pi_1\cdot\pi_1,\pi_2)^\circ$. This premise is trivially proved. In the second premise:

$$Opt((b\_exp_1, b\_exp_2)^\circ \Rightarrow b\_exp)$$

b_exp_1 is, again, bound to: $!\cdot(\pi_1\cdot\pi_1,\pi_2)^\circ$ and b_exp_2 is bound to: π_2. In this premise, Opt recursively descends into the constituent functions of the alltup. The code for b_exp_1: $!\cdot(\pi_1\cdot\pi_1,\pi_2)^\circ$ eventually matches rule 24 (select-introduction) of the *IOpt* rule-set, producing the code:

$$\mathsf{select}\cdot(\pi_1\cdot\pi_1,\pi_2)^\circ$$

the code for b_exp_2: π_2 is mapped, by rules for core processing of non-index functions, to itself producing:

$$\pi_2$$

these two sub-results are combined and put into context by post processing to form the result:

$$\mathsf{zip}\cdot(\mathsf{select}\cdot(\pi_1\cdot\pi_1,\pi_2)^\circ,\pi_2)^\circ\cdot(\mathsf{id},\mathsf{iota}\cdot\#\cdot\pi_1)^\circ$$

which corresponds to the b_exp at the end of the second premise shown above. When integrated back into the code from figure 65 the code in figure 66 is produced. The

$$\underline{((!) * \cdot \mathsf{zip} \cdot (\mathsf{select} \cdot (\pi_1 \cdot \pi_1, \pi_2)^\circ, \pi_2)^\circ \cdot (\mathsf{id}, \mathsf{iota} \cdot \# \cdot \pi_1)^\circ)} * \mathsf{distl} \cdot (\mathsf{id}, \mathsf{iota} \cdot \#)^\circ$$

Figure 66. The code from figure 65 after the application of rule 26 (catch-all). Underlined code corresponds to the underlined code from figure 65.

$$((! \cdot (! \cdot (\pi_1 \cdot \pi_1, \underline{\pi_2})^\circ, \pi_2 \cdot \pi_1)^\circ) * \cdot \mathsf{distl} \cdot (\mathsf{id}, \mathsf{iota} \cdot \# \cdot \pi_1)^\circ) * \cdot \mathsf{distl} \cdot (\mathsf{id}, \mathsf{iota} \cdot \#)^\circ$$

Figure 67. A program that transposes its input vector. The π_2 function in the inner indexing function is underlined.

underlined code is the new code that has been produced by rule 26. The program in figure 66 is substantially more efficient than the translator code code in figure 65 because it only performs distribution of the input vector on one level.

Second example: vector transposition Figure 67 shows a nested map-translation with a nested indexing function. This code transposes any square-shaped input vector given to it by distributing copies of the input to invocations of a nested map-body and then using a nested indexing function to extract the appropriate values. Like other translator code we have seen thus far, this code is not efficient. Unfortunately, the inner indexing function has a π_2 function as one of its most-upstream functions, which makes the inner map-translation applicable only to rule 26, the catch-all rule.

Figure 68 shows the code in figure 67 after the application of rule 26 (catch-all). The function:

$$\mathsf{select} \cdot (\pi_1 \cdot \pi_1, \pi_2)^\circ$$

is generated, as per the previous example, from the inner indexing function:

$$! \cdot (\pi_1 \cdot \pi_1, \pi_2)^\circ$$

The function:

$$((!) * \cdot \mathsf{zip} \cdot (\mathsf{select} \cdot (\pi_1 \cdot \pi_1, \pi_2)^\circ, \mathsf{repeat} \cdot (\pi_2 \cdot \pi_1, \# \cdot \pi_2)^\circ)^\circ \cdot (\mathsf{id}, \mathsf{iota} \cdot \# \cdot \pi_1)^\circ) * \cdot \mathsf{distl} \cdot (\mathsf{id}, \mathsf{iota} \cdot \# \cdot \pi_1)^\circ$$

Figure 68. Code from figure 67 after optimisation using rule 26 (catch-all).

$$\mathsf{repeat} \cdot (\pi_2 \cdot \pi_1, \# \cdot \pi_2)^\circ$$

is derived from the index-generating-function:

$$\pi_2 \cdot \pi_1$$

by core-optimiser rules for non indexing-functions. The overall efficiency of the code in figure 68 is much improved over its corresponding translator code but it is still far from optimal. In particular, the code still distributes the entire input vector over the outer dimension of the input vector. An optimal or near-optimal version of the program would avoid the cost of this distribution by *transposing* the input vector and then selecting out the appropriate elements. We next explain the mechanism by which the optimiser introduces such transposition.

Reorientation of Vectors The code in figure 67 is close to being a very good candidate for optimisation by the rules of *IOpt*. If the indexing function were:

$$! \cdot (! \cdot (\pi_1 \cdot \pi_1, \pi_2 \cdot \pi_1)^\circ, \pi_2)^\circ$$

instead of:

$$! \cdot (! \cdot (\pi_1 \cdot \pi_1, \pi_2)^\circ, \pi_2 \cdot \pi_1)^\circ$$

we would have the same code as that shown in figure 56 which, we know, can be very effectively optimised without resorting to the, less effective, catch-all rule (rule 26). This observation points the way to an effective strategy: reorient the vector that is accessed by the indexing function and simultaneously reorient the nesting of the indexing functions to compensate. In the example above, this reorientation must take the form of a $\mathsf{transpose}$ function where[25]:

$$\forall x, \forall y, (\mathsf{tranpose}(a)!x)!y = (a!y)!x$$

Applying this reorientation strategy to our current example produces the following transformation:

$$! \cdot (! \cdot (\pi_1 \cdot \pi_1, \pi_2)^\circ, \pi_2 \cdot \pi_1)^\circ \Rightarrow ! \cdot (! \cdot (\mathsf{transpose} \cdot \pi_1 \cdot \pi_1, \pi_2 \cdot \pi_1)^\circ, \pi_2)^\circ$$

Figure 69 shows the effect of this transformation on the program given in figure 67. The index function is now amenable to optimisation by applying the rules of *IOpt* used for the program in figure 56. The result of applying these rules to the program

$$((! \cdot (! \cdot (\mathsf{transpose} \cdot \pi_1 \cdot \pi_1, \pi_2 \cdot \pi_1)^\circ, \pi_2)^\circ) * \cdot \mathsf{distl} \cdot (\mathsf{id}, \mathsf{iota} \cdot \# \cdot \pi_1)^\circ) * \cdot \mathsf{distl} \cdot (\mathsf{id}, \mathsf{iota} \cdot \#)^\circ)^\circ$$

Figure 69. Program from figure 67 after inserting a function to transform the input vector and rearranging the index functions to compensate.

$$(\mathsf{select} \cdot (\pi_2, \mathsf{iota} \cdot \pi_1)^\circ) * \cdot \mathsf{distl} \cdot (\# \cdot \mathsf{id}, \mathsf{select} \cdot (\mathsf{transpose} \cdot \mathsf{id}, \mathsf{iota} \cdot \#)^\circ)^\circ$$

Figure 70. Fully optimised version of the program shown in figure 69.

in figure 69 is the program in figure 70. This program is of the same order of efficiency as optimised nested **maps** without transposition. The **transpose** primitive, if properly implemented on a sequential machine, is $O(n)$ cost where n is the total number of elements in the nested input vector.

The insertion of **transpose** can be used to increase the effectiveness of optimisation wherever it is possible to increase the degree to which index-generators in a nested indexing functions are *sorted*. A precise description of what it means for index-generators to be sorted is given in appendix C but some insight can be gained by considering how to use **transpose** to help optimise more-deeply nested indexing functions.

Coping with more levels of indexing The reorientation strategy described above swaps the indices of adjacent levels of nested index functions to achieve an index order more amenable to further optimisation. This strategy naturally extends to more deeply nested index functions in the same way that swapping of adjacent elements of a list can be used to implement a bubble-sort. For example, figure 71 part (a) shows a triple-nested indexing function prior to reorientation and part (b) shows the same function after reorientation. The sequence of **transpose** functions in part (b) performs the series of re-orientations necessary to swap the indices to a sorted order. Note that the sorted order has π_2 which references the range-generator of the innermost **map** translation as the outermost index, followed by $\pi_2 \cdot \pi_1$ and $\pi_2 \cdot \pi_1 \cdot \pi_1$. If we define the following symbolic mapping:

$$\{L_2 \mapsto \pi_2 \cdot \pi_1 \cdot \pi_1, L_1 \mapsto \pi_2 \cdot \pi_1, L_0 \mapsto \pi_2, V \mapsto \pi_1 \cdot \pi_1 \cdot \pi_1\}$$

[25]Note where x and y reference an element outside the bounds of the nested vector $\perp$ is returned.

$$! \cdot (! \cdot (! \cdot (\pi_1 \cdot \pi_1 \cdot \pi_1, \pi_2)^\circ, \pi_2 \cdot \pi_1)^\circ, \pi_2 \cdot \pi_1 \cdot \pi_1)^\circ$$

(a)

$$! \cdot (! \cdot (! \cdot (\mathsf{tranpose} \cdot (\mathsf{transpose}) * \cdot \mathsf{transpose} \cdot \pi_1 \cdot \pi_1 \cdot \pi_1, \pi_2 \cdot \pi_1 \cdot \pi_1)^\circ, \pi_2 \cdot \pi_1)^\circ, \pi_2)^\circ$$

(b)

Figure 71. A triple-nested index function (a) that can be reoriented to a form more amenable to optimisation (b).

$$\begin{array}{lr}
! \cdot (! \cdot (! \cdot (V, L_0)^\circ, L_1)^\circ, L_2)^\circ & \Rightarrow \\
! \cdot (! \cdot (! \cdot (\mathsf{transpose} \cdot V, L_0)^\circ, L_2)^\circ, L_1)^\circ & \Rightarrow \\
! \cdot (! \cdot (! \cdot ((\mathsf{transpose}) * \cdot \mathsf{transpose} \cdot V, L_2)^\circ, L_0)^\circ, L_1)^\circ & \Rightarrow \\
! \cdot (! \cdot (! \cdot (\mathsf{transpose} \cdot (\mathsf{transpose}) * \cdot \mathsf{transpose} \cdot V, L_2)^\circ, L_1)^\circ, L_0)^\circ &
\end{array}$$

Figure 72. The transformation steps used to reorient the vector accessed by the function in in figure 71(a) under the mapping $\{L_2 \mapsto \pi_2 \cdot \pi_1 \cdot \pi_1, L_1 \mapsto \pi_2 \cdot \pi_1, L_0 \mapsto \pi_2, V \mapsto \pi_1 \cdot \pi_1 \cdot \pi_1\}$.

then a sequence of steps by which we arrive at the code in figure 71(b) is shown in figure 72. The analogy between the application of the transformations and sorting is apparent, with each step above corresponding to a swap of dimensions. Note that the second swap transposes inner dimensions, hence the embedding of transpose in a map function of its own.

Note that because transpose is limited to swapping adjacent dimensions of a nested vector we had to use transpose three times to achieve the desired orientation. If we assume the existence of a more powerful transpose primitive, able to swap a pair of dimensions an arbitrary distance apart, we can sort the index-generators in fewer steps. Figure 73 shows the transformation of the nested index function from figure 71(a) to a sorted format using a transpose primitive parameterised with the dimensions that need to be swapped. Note that 0 denotes the outermost dimension and 2 denotes the dimension two levels of nesting down. The order in which these appear in the subscripted brackets does not affect the semantics of the transpose operation. Note that this parameterised form of transpose still performs a single swap. Where more than one swap is required to sort the dimensions of the array then more than one application of the parameterised transpose is still required.

$$\begin{array}{l} ! \cdot (! \cdot (! \cdot (\pi_1 \cdot \pi_1 \cdot \pi_1, \pi_2)^\circ, \pi_2 \cdot \pi_1)^\circ, \pi_2 \cdot \pi_1 \cdot \pi_1)^\circ \qquad \Rightarrow \\ ! \cdot (! \cdot (! \cdot (\mathsf{transpose}_{(0,2)} \cdot \pi_1 \cdot \pi_1 \cdot \pi_1, \pi_2 \cdot \pi_1 \cdot \pi_1)^\circ, \pi_2 \cdot \pi_1)^\circ, \pi_2)^\circ \end{array}$$

Figure 73. A transformation from the code in figure 71(a) to code amenable to further optimisation using a parameterised transpose primitive.

The implementation of transpose In our discussion to date we have employed the transpose primitive without regard to the details of its implementation. This implementation is non-trivial for nested vectors, because sub-vectors may be of different lengths. The implementation of the parametric version of transpose is even more complex. While detailed knowledge of how transpose works is not essential for an understanding of the foregoing, it is important that a plausible schema for its implementation exists.

In brief, our strategy for implementation recognises that the transpose of irregular nested vectors can, if not carefully handled, lead to the violation of simple identities such as:

$$transpose_{(01)} \cdot \mathsf{transpose}_{(0,1)}\ x = x$$

The key to maintaining correctness, and thus preserve such identities, is to keep, and abide by, information relating to the orginal shape of the nested vector.

There are many plausible schemas for preserving shape information[26]. We have defined a prototype implementation of parametric transpose where shape information is kept by ecoding index information along with values and maintaining sentinel values.

The reader is referred to appendix B for details relating to the semantics of parametric transpose as applied to nested vectors along with code for the prototype implementation, written in Scheme.

Limits of re-orientation using transpose It is not always possible to use transpose to avoid the application of the catch-all rule of *IOpt*. For example, figure 74 shows a program where, no matter how the index-generating functions are swapped

[26]Such preservation is central to defining parallel computation across nested data using structure-flattening. Examples of implementations exploiting such flattening are NESL[22, 23] and, more recently, Nepal [30, 29]

$$((! \cdot (! \cdot (\pi_1 \cdot \pi_1, + \cdot (\pi_2, \pi_2 \cdot \pi_1)^\circ)^\circ, + \cdot (\pi_2, \pi_2 \cdot \pi_1)^\circ)^\circ) * \cdot \mathsf{distl} \cdot (\mathsf{id}, \mathsf{iota} \cdot 4)^\circ) * \mathsf{distl} \cdot (\mathsf{id}, \mathsf{iota} \cdot 4)^\circ$$

Figure 74. A program with a nested index function where reorientation would be ineffective in avoiding the application of the catch-all rule in further optimisation.

$$((! \cdot A, ! \cdot B)^\circ) * \cdot \mathsf{distl} \cdot (V, R)^\circ$$
$$(a)$$

$$\mathsf{zip} \cdot (A', B')^\circ \cdot (V, R)^\circ$$
$$(b)$$

Figure 75. An archetypal schema for a map-translation with multiple indexing functions (part (a)) and the same schema after core-optimisation has been applied (part (b)).

around, a π_2 function will always appear as the most-upstream function in an inner-indexing function. In short, no amount of swapping of index-generators will get all of the π_2 functions to the outside. This property is also is also manifest in the program, shown earlier, in figure 65. Under these circumstances the optimiser's rules for reorientation will sort index-generators as well as possible but there will still, inevitably, be some distribution of data remaining in optimised code[27].

A precise specification of indexing functions that are sorted or amenable to sorting is given in appendix C.

Handling Multiple Indexing Functions The following describes work that is done by the core optimisation process and the post-processor when applied to map-bodies containing more than one indexing function. We describe core processing first.

Core processing of multiple indexing functions Figure 75(a) shows an archetypal schema for a map-translation containing multiple indexing functions. Part (b) shows the same code after the application of core optimisation. The code in A'

[27]Note that the presence of distribution in the code does not imply that the code is non-optimal. For some problems the optimal solution will involve distribution.

$$((! \cdot (\pi_1, 0)^\circ, ! \cdot (\pi_1, 1)^\circ)^\circ) * \cdot \mathsf{distl} \cdot (\mathsf{id}, \mathsf{iota} \cdot \#\mathsf{id})^\circ$$

Figure 76. map-translation containing multiple index functions with constant indices.

$$\mathsf{zip} \cdot (\mathsf{repeat} \cdot (! \cdot (\pi_1, 0)^\circ, \# \cdot \pi_2)^\circ, \mathsf{repeat} \cdot (! \cdot (\pi_1, 1)^\circ, \# \cdot \pi_2)^\circ)^\circ \cdot (\mathsf{id}, \mathsf{iota} \cdot \#\mathsf{id})^\circ$$

Figure 77. Optimised map-translation of code from figure 76 prior to post-processing.

is the code that would result from core optimisation of $! \cdot A$ as if it were alone in the map body. That is A' is the A' in:

$$(! \cdot A) * \cdot \mathsf{distl} \cdot (V, R)^\circ \overset{\Rightarrow}{_{IOpt}} A' \cdot (V, R)^\circ$$

Where the $\overset{\Rightarrow}{_{IOpt}}$ transformation denotes the application of the *IOpt* rule set to the code to its left. In figure 75 part (b) the pair of equal-length vectors[28] produced by $(A', B')^\circ$ is combined into a single vector of pairs using zip.

Figure 75 is adequate for explaining how core-optimisation processes map-bodies containing pairs of indexing functions. In the general case, arbitrarily many index operations can appear in alltup functions of varying arity. The core-optimisation rules are written to handle all such cases by inserting the appropriate combinations of alltup, addressing and zip functions.

Post-processing of multiple indexing functions Sometimes the application of core-processing leaves scope for further improvement to the code. For example figure 76 shows a map-translation containing multiple index functions with constant indices. Figure 77 shows the same code after the application of core processing. The repeat functions copy their respective arguments the same number of times which makes them amenable to consolidation. Figure 78 shows the program from figure 77 after the repeat functions have been consolidated by post-processing.

As a further example, figure 79 shows an unoptimised map-translation containing both constant and non-constant index generators. Core-processing produces the code shown in figure 80. Post-processing is able to remove the repeat, select and zip

[28]These vectors are always the same length as the vector produced by the range-generator R.

$$\mathsf{repeat} \cdot ((! \cdot (\pi_1, 0)^\circ, ! \cdot (\pi_1, 1)^\circ)^\circ, \# \cdot \pi_2)^\circ \cdot (\mathsf{id}, \mathsf{iota} \cdot \#\mathsf{id})^\circ$$

Figure 78. Post-processed version of the optimised code shown in figure 77.

$$(! \cdot (\pi_1, \pi_2)^\circ, ! \cdot (\pi_1, 0)^\circ) * \cdot \mathsf{distl} \cdot (\mathsf{id}, \mathsf{iota} \cdot \#\mathsf{id})^\circ$$

Figure 79. Unoptimised map-translation containing an index function with a non-constant index-generator and an index function with a constant index-generator.

functions from this code by taking advantage of the fact that the output of both select and repeat are of the same length so equivalent program semantics can be attained by distributing the first parameter of repeat over the output of select. Figure 81 shows the result of applying such post-processing to the current example. The underlined code is used to bring the tuples in the output vector back into their original order.

This concludes our description of the vector optimisation process for handling map-bodies with multiple-indexing functions. Next we describe how the processing of conditional functions containing indexing functions is done.

Handling Indexing Functions in Conditional Functions Often map-bodies will contain if functions. In turn, these if functions can contain index functions that might be amenable to optimisation. Figure 82 shows a map-translation that interleaves the first half of an input vector with its second half. An if function in the map-body is used to achieve this. This if function can be paraphrased:

$$\mathsf{if}(\pi_2\ \mathit{is\_even}, \mathit{access\_first\_half}\ \mathit{of}\ \pi_1, \mathit{access\_second\_half}\ \mathit{of}\ \pi_1)$$

the function *access_first_half* corresponds to a simple select function:

$$\mathsf{select} \cdot (\pi_1, \mathit{first\_half\_of\_indices})^\circ$$

Likewise, the function *access_second_half* corresponds to the select function

$$\mathsf{zip} \cdot (\mathsf{select} \cdot (\pi_1, \pi_2)^\circ, \mathsf{repeat} \cdot (! \cdot (\pi_1, 0)^\circ, \# \cdot \pi_2)^\circ)^\circ \cdot (\mathsf{id}, \mathsf{iota} \cdot \#\mathsf{id})^\circ$$

Figure 80. Core-optimised version of the code shown in figure 79.

$$\underline{((\pi_2,\pi_1)^\circ)*} \cdot \mathsf{distl} \cdot (! \cdot (\pi_1, 0)^\circ, \mathsf{select} \cdot (\pi_1,\pi_2)^\circ)^\circ \cdot (\mathsf{id}, \mathsf{iota} \cdot \#\mathsf{id})^\circ$$

Figure 81. Post-processed version of the code shown in figure 79. The underlined code brings tuples in the output vector back into their original order.

$$\begin{aligned}
(\mathsf{if}(& \overbrace{= \cdot(\mathsf{mod} \cdot (\pi_2, 2)^\circ, 0)^\circ}^{\pi_2\ is\_even}, \\
& \overbrace{! \cdot (\pi_1, \div \cdot (\pi_2, 2)^\circ)^\circ}^{access\_first\_half}, \\
& \overbrace{! \cdot (\pi_1, + \cdot (\div \cdot (\# \cdot \pi_1, 2)^\circ, \div \cdot (\pi_2, 2)^\circ)^\circ)^\circ}^{access\_second\_half})) * \cdot \\
\mathsf{distl} & \cdot (\mathsf{id}, \mathsf{iota} \cdot \# \cdot \mathsf{id})^\circ
\end{aligned}$$

Figure 82. map-translation containing an if function that interleaves the top and bottom halves of an input vector. The first, second and third parameters to the higher-order if functions are marked with over-braces.

$$\mathsf{select} \cdot (\pi_1, second\_half\_of\_indices)^\circ$$

Where *first_half_of_indices* and *second_half_of_indices* are functions for producing the first and second half of the range-generator respectively.

These two select functions indicate that there is scope for vector-optimisation, buried inside the if function. However, a way still needs to be found to model the way the if function, in the example, alternates between values from the first and second half of the vector. The vector-optimiser achieves this alternation using two different functions:

- A primitive function called mask that uses the boolean results of the predicate-part of the if function to filter out the indices not relevant to each select.
- A primitive function called priffle that interleaves the result of the select's using the boolean results of the predicate of the if

Figure 83 shows the optimised version of the map-translation from figure 82 containing the new primitives. The optimised code, corresponding to the original parameters to the if function are marked with over-braces. The primes on the labels for this code indicate that this code has been vector-optimised. The remainder of the code, with the

$$\begin{array}{l}
\mathsf{priffle}\cdot \\
(\pi_2, \\
\overbrace{\mathsf{select}\cdot(\pi_1,\div\cdot(\pi_2,2)^\circ)^\circ}^{access\_first\_half'}\cdot(\pi_1\cdot\pi_1,\mathsf{mask}\cdot(\pi_2\cdot\pi_1,\pi_2)^\circ)^\circ, \\
\overbrace{\mathsf{select}\cdot(\pi_1,+\cdot(\div\cdot(\#\cdot\pi_1,2)^\circ,\div\cdot(\pi_2,2)^\circ)^\circ)^\circ}^{access\_second\_half'}\cdot(\pi_1\cdot\pi_1,\mathsf{mask}\cdot(\pi_2\cdot\pi_1,(\neg)*\cdot\pi_2)^\circ)^\circ)^\circ\cdot \\
\qquad\qquad\qquad\qquad(\mathsf{id},\overbrace{(=\cdot(\mathsf{mod}\cdot(\pi_2,2)^\circ,0)^\circ)*\cdot\pi_2}^{\pi_2\,is\_even'})^\circ\cdot(\mathsf{id},\mathsf{iota}\cdot\#\cdot\mathsf{id})^\circ
\end{array}$$

Figure 83. Optimised version of the map-translation from figure 82 with the optimised parameter functions to the original if marked with over-braces. All of the remaining code, except the most upstream alltup function requires no customisation. It does not vary from program to program.

exception of the most upstream alltup function, is a standard template for optimising the if function. This template does not vary from program to program.

The role of the mask and priffle functions are central to the optimised code. The mask function takes a vector of values and vector of booleans for parameters and filters out all of the values of the original vector for which the corresponding element of the boolean vector is *false*. mask is defined:

$$\begin{array}{lcl}
\mathsf{mask}\,(([x]+\!\!+\,xs),([true]+\!\!+\,ys)) & = & [x]+\!\!+\,\mathsf{mask}\,(xs,ys) \\
\mathsf{mask}\,(([x]+\!\!+\,xs),([false]+\!\!+\,ys)) & = & \mathsf{mask}\,(xs,ys) \\
\mathsf{mask}\,([\,],[\,]) & = & [\,]
\end{array}$$

In the optimised code, in figure 83 mask uses the vector of boolean values produced by the π_2 *is_even'* function to filter out the index values for which the original predicate function of the if function is *false* (for the code accessing the first half of the input vector) or *true* (for the code accessing the second half of the input vector). The mask functions, in combination with the select functions, extract the appropriate parts of the original input vector.

Once the consequent and alternative functions, corresponding to *access_first_half'* and *access_second_half'* respectively in the example in figure 83, have produced their results, we have two vectors whose combined length is the same as the original input vector. These vectors need to be interleaved so that the order of results produced by the original if function is preserved. The interleaving is carried out by the priffle function.

priffle (the name is a contraction of predicate-riffle[29]) takes, as input, a vector of boolean values and two other vectors. priffle selects items from each of its second two vectors according to whether the next value in the boolean vector is *true* or *false*. In other words priffle is a merging operation with a predicate parameter to guide it. priffle is defined:

$$\begin{aligned}
\mathsf{priffle}\,([\mathit{true}]\mathbin{+\!\!+} xs, [y]\mathbin{+\!\!+} ys, zs) &= [y]\mathbin{+\!\!+} \mathsf{priffle}(xs, ys, zs)\\
\mathsf{priffle}\,([\mathit{false}]\mathbin{+\!\!+} xs, ys, [z]\mathbin{+\!\!+} zs) &= [z]\mathbin{+\!\!+} \mathsf{priffle}(xs, ys, zs)\\
\mathsf{priffle}\,([\,],[\,],[\,]) = [\,]
\end{aligned}$$

Note that the final case assumes that all vectors are exhausted at the same time. The context in which the mask and priffle are embedded, by the vector optimiser, ensures that this is the case.

It should be noted that neither priffle or mask or the way in which they are used here are unique. Blelloch uses similar techniques in the implementation of NESL[22, 23]. Prins and Palmer[112] describe `restrict` and `combine` operations in the implementation of *proteus*, corresponding to mask and priffle respectively. Finally, the role of two mask functions could be replaced by `partition` from the standard *List* library of Haskell[30].

This concludes our discussion of the optimisation of if functions in map-bodies and also our discussion of vector optimisation of indexing-functions in general. Next, we briefly discuss the optimisation of other functions found in the map-body.

5.4.2.4 Processing Non-Indexing Functions inside Map

Given a map translation:

$$(\mathit{bexp}) * \cdot \mathsf{distl} \cdot (V, R)^{\circ}$$

The purpose of vector optimisation is to move indexing and length functions inside *bexp* as far upstream as possible. The rules we have just described are designed to fulfill this purpose. However, these rules can rarely work in isolation because, *bexp* will, often, contain other functions beside index and length functions. In fact, index

[29]Riffle is a term borrowed from Ruby [87]. The original meaning describes the way in which two half-decks of cards can be combined, quickly, by skillfully interleaving cards from the half-decks.

[30]The replacement of the two mask's by a single partition is an option well worth exploring in a future version of this implementation.

and length functions are commonly embedded in other functions such as alltup. Rules must be in place to handle these functions to ensure that:

1. the vector-optimised code is left in a consistent state with the same semantics as the original translation.

2. no opportunities for further vector optimisation are foregone. That is, any code that is amenable to further optimisation is positioned so that it is *exposed* to further optimisation.

These roles are obviously important and the rules that we defined to fulfill these roles are non-trivial. However, the details of these rules are not central to our discussion of the vector optimiser and we omit them from this report. At this point it is sufficient to note that such rules are required and they have the roles enumerated above.

This concludes our description of the vector optimisation of map-translations. Next we briefly describe the vector optimisation of other functions.

5.4.3 Optimising non-map functions

Even though map functions are the focus of most of the work of the vector optimiser there is still important work to be done involving other functions.

The remaining part of the vector optimiser carries out two major tasks:

- Constructing, maintaining, and propagating the wish-list.

- Specialised handling for the translation of the Adl `reduce` and `scan` functions.

We discuss these tasks in turn.

5.4.3.1 Maintenance of the wish-list

All of the top-level rules for the vector-optimiser produce a composed function of the form:

$$C \cdot W$$

where C is the code that is already optimised and W is a wish-list.

Together, the top-level rules ensure that:

- Index and length functions are inserted into the wish-list and they stay in the wish-list until they are either eliminated or can be pushed no further upstream.

- Any code downstream of any index or length function remains with the optimised code.

As a consequence of these two rules the wish-list W can take one of two forms. The first form:

$$W = \mathsf{id}$$

occurs when there are no index or length functions awaiting further processing. The second form:

$$W = \mathit{bexp}\ \textbf{\textit{where}}\ \mathit{bexp} \neq \mathsf{id} \wedge (\mathit{Subterm}(\mathit{bexp}, !) \vee \mathit{Subterm}(\mathit{bexp}, \#))$$

occurs when there are index or length functions awaiting processing (the predicate $\mathit{Subterm}(f, x)$ is true if x is a sub-term of f).

Each vector optimiser rule described below is responsible for creating and maintaining wish lists corresponding to one of these two forms. For clarity, in each of the following rules the output code that corresponds to the new wish-list W is underlined.

Constant functions, Ordinary operators and singleton address functions Constant functions require some input to trigger their evaluation but their result is not dependent on the value of their input. Constant functions are first-order so they cannot contain any index or length functions, or any other functions for that matter. The rule for constant functions is

$$\mathsf{K} \Rightarrow \mathsf{K} \cdot \underline{\mathsf{id}}$$

The constant function K is the section containing the already-optimised code. The rules for non-index, non-length, operators

$$\mathsf{Op} \Rightarrow \mathsf{Op} \cdot \underline{\mathsf{id}} \qquad (\mathsf{Op} \neq ! \wedge \mathsf{Op} \neq \#)$$

and singleton[31] address functions

$$^m\pi_n \Rightarrow ^m \pi_n \cdot \underline{\mathsf{id}}$$

are also very similar. In all of these cases an id wish-list is formed.

[31]A singleton function is a function f is any function that is not a composition. That is $\mathit{Singleton}(f) \iff \forall f, g \neg \mathit{Unify}(f, \mathsf{b\_comp}(g, h))$. Where the predicate $\mathit{Unify}(s, t)$ is true if s can be structurally unified with t.

Index and length functions The rules for handling index and length functions are

$$\frac{bexp \Rightarrow bexp' \cdot bexp''}{! \cdot bexp \Rightarrow \mathsf{id} \cdot \underline{(! \cdot bexp' \cdot bexp'')}}$$

and

$$\frac{bexp \Rightarrow bexp' \cdot bexp''}{\# \cdot bexp \Rightarrow \mathsf{id} \cdot \underline{(\# \cdot bexp' \cdot bexp'')}}$$

respectively. The new wish-list contains either an index or length function composed with $bexp' \cdot bexp''$ output of the vector optimiser for the code immediately upstream. The wish-lists produced by these rules will serve as vehicles for transporting the index and length functions further upstream in case an opportunity to optimise them further arises.

Composition sequences The top-level rules of the vector optimiser are structured around code-fragments produced by the translator. As such the treatment of composed sequences of functions is distributed among specialised rules to handle these fragments. However, the vector optimiser's approach to handling composition sequences can be described by four archetypal rules. These rules are shown in figure 84 Note again that the wish-lists are underlined. Rule 27 handles the cases when the processing of both the downstream $bexp_1$ and upstream $bexp_2$ functions produces id wish-lists. These combine to form a new id wish-list. Rule 28 applies when the upstream function $bexp_2$ produces a non-id wish-list: $bexp_2''$ which is also the wish-list returned in the conclusion of the rule.

Rule 27 applies when the downstream function $bexp_1$ produces a non-identity wish-list $bexp_1''$. This wish-list is then combined with the non-identity part of the result of processing $bexp_2$ to form the new wish-list:

$$bexp_1'' \cdot bexp_2'$$

Note that, even though $bexp_2'$ contains no index or length functions it must be included in the wish-list because these functions are contained in the $bexp_1''$ and there is no semantically sound way, without further processing, to *push* $bexp_1''$ through $bexp_2'$. This behaviour of upstream functions being co-opted into the wish-list is quite typical. This behaviour is also necessary because, as we have seen during the optimisation of map, the upstream code that supplies the arguments of length and index operations needs to be analysed as part of the optimisation process for these functions.

Set *VecOpt* **is** $\boxed{b\_exp \Rightarrow b\_exp}$

Both identity wish-lists

$$\begin{array}{c} bexp_1 \Rightarrow bexp'_1 \cdot \underline{\mathsf{id}} \\ bexp_2 \Rightarrow bexp'_2 \cdot \underline{\mathsf{id}} \\ \hline bexp_1 \cdot bexp_2 \Rightarrow bexp'_1 \cdot bexp'_2 \cdot \underline{\mathsf{id}} \end{array} \qquad (27)$$

Second wish-list is non-identity

$$\begin{array}{c} bexp_1 \Rightarrow bexp'_1 \cdot \underline{\mathsf{id}} \\ bexp_2 \Rightarrow bexp'_2 \cdot \underline{bexp''_2} \\ \hline bexp_1 \cdot bexp_2 \Rightarrow bexp'_1 \cdot bexp'_2 \cdot \underline{bexp''_2} \end{array} \quad bexp''_2 \neq \mathsf{id} \qquad (28)$$

First wish-list is non-identity

$$\begin{array}{c} bexp_1 \Rightarrow bexp'_1 \cdot \underline{bexp''_1} \\ bexp_2 \Rightarrow bexp'_2 \cdot \underline{\mathsf{id}} \\ \hline bexp_1 \cdot bexp_2 \Rightarrow bexp'_1 \cdot \underline{bexp''_1 \cdot bexp'_2} \end{array} \quad bexp''_1 \neq \mathsf{id} \qquad (29)$$

Both wish-lists are non-identity

$$\begin{array}{c} bexp_1 \Rightarrow bexp'_1 \cdot \underline{bexp''_1} \\ bexp_2 \Rightarrow bexp'_2 \cdot \underline{bexp''_2} \\ \hline bexp_1 \cdot bexp_2 \Rightarrow \mathsf{id} \cdot \underline{bexp'_1 \cdot bexp''_1 \cdot bexp'_2 \cdot bexp''_2} \end{array} \quad \begin{array}{c} bexp''_1 \neq \mathsf{id} \wedge \\ bexp''_2 \neq \mathsf{id} \end{array} \qquad (30)$$

Figure 84. Archetypal vector optimiser rules for handling composed functions

The last rule in figure 84 is applied when both sets of wish-lists are non-identity functions. In this case all code resulting from vector optimisation of $bexp_1$ and $bexp_2$ goes into the wish-list.

Note that the rules in figure 84 are effective in *building up* the wish-list. The counterpart to this activity, the *destruction* of the wish-list, took place inside the the *Process_Map* rule set described earlier. This completes the description the handling of composed functions next we consider the handling of alltups.

Alltup functions There are two rules for alltup functions. The first rule:

$$\frac{\begin{array}{c} bexp_1 \Rightarrow bexp'_1 \cdot \underline{\mathsf{id}} \\ \dots \\ bexp_n \Rightarrow bexp'_n \cdot \underline{\mathsf{id}} \end{array}}{(bexp_1, \dots, bexp_n)^\circ \Rightarrow (bexp'_1, \dots, bexp'_n)^\circ \cdot \underline{\mathsf{id}}}$$

applies when all of the functions inside the alltup produce id wish-lists. In this case the wish-lists can be consolidated into a single id function.

The second rule:

$$\frac{\begin{array}{c} bexp_1 \Rightarrow bexp'_1 \cdot \underline{bexp''_1} \\ \dots \\ bexp_n \Rightarrow bexp'_n \cdot \underline{bexp''_n} \end{array}}{(bexp_1, \dots, bexp_n)^\circ \Rightarrow (bexp'_1 \cdot^n \pi_1, \dots, bexp'_n \cdot^n \pi_n)^\circ \cdot \underline{(bexp''_1, \dots, bexp''_n)^\circ}} \quad \begin{array}{c} bexp''_1 \neq \mathsf{id} \vee \\ \dots \\ bexp''_n \neq \mathsf{id} \end{array}$$

applies when at least one of the functions in the alltup produces a non-id wish-list. In this case all wish-lists are grouped into an alltup function. Addressing functions link each function in the downstream alltup function to its corresponding wish-list in the upstream alltup.

Other functions The examples shown so far, illustrate most of the techniques employed by the top-level rules of the vector optimiser. The rule for allvec functions is very similar to the rule for alltup functions.

The rules for if and while are very conservative; if any of the functions they encapsulate produces a non-identity wish-list then all code goes in the wish-list, leaving just an id function as optimised code.

The rules for the various types of reduce and scan have similarly conservative strategies. However, these constructs also require additional, specialised processing and we discuss this next.

5.4.3.2 Handling reduce and scan translations

As mentioned in the previous chapter, the translation of `reduce` and `scan` functions produces very detailed code. A translation for `reduce` (a reduce-translation) takes the form shown in figure 85, where R is the range-generator for the input vector and Z is the code producing the value that is the left and right identity for the binary function $\oplus$. The if function determines if the input vector is empty. If it is empty, all data except for the input vector is directed to the function Z which then produces the output for the entire reduce-translation. If the vector is non-empty then the

$$\mathsf{if}(\neq \cdot(0, \# \cdot \pi_2)^\circ, < \textit{detailed-code} > \oplus < \textit{other-detailed-code} > / \cdot \mathsf{distl}, Z \cdot \pi_1) \cdot (\mathsf{id}, R)^\circ$$

Figure 85. The basic form of code produced by the translator for `reduce` functions.

$$< \textit{detailed-code} > \oplus < \textit{other-detailed-code} > /_Z \cdot \mathsf{distl} \cdot (\mathsf{id}, R)^\circ$$

Figure 86. A reduce-translation specialised with the knowledge that Z is a constant function.

input vector, produced by R is distributed over the other values in scope. The vector resulting from this distribution is then given to a **reduce** function with the binary $\oplus$ function embedded in detailed code to ensure that $\oplus$ receives an input tuple that is wrapped up with all of the global data that it may require.

Almost all of this heavy infrastructure is there to cater for two relatively rare cases.

1. The function Z being dependent on one of the values in scope.
2. The function $\oplus$ being dependent on global values; that is values other than its immediate arguments.

If either of these cases is *not* true then substantial optimisations can be made. Addressing the first case, if Z is a constant function, and thus *not* dependent on any particular value then the code in figure 85 is simplified to the code in figure 86. Note that the function:

$$\mathsf{distl} \cdot (\mathsf{id}, R)^\circ$$

in figure 86 will produce an empty vector whenever R produces an empty vector. An empty input vector would leave the function Z with no means of accessing any values in scope. However, because Z is a constant function as we assume in this discussion, the modified code still works.

Now, addressing the second case - if $\oplus$ is *not* dependent on any values in global scope then the code in figure 86 can be further simplified to the code in figure 87.

This code lacks the **distl** function of the previous versions and any of the infrastructure for carting around values in global scope and is very much more efficient

$$\oplus/_Z$$

Figure 87. A reduce-translation specialised with the knowledge that Z is a constant function *and* that $\oplus$ depends only on its immediate operands.

than the original reduce-translation. In most of our experiments the form shown in figure 87 is the form ultimately produced by the vector optimiser.

Note that the optimiser must analyse the code in $\oplus$ in order to determine what values it depends upon. This analysis requires both vector and tuple optimisation of $\oplus$ to eliminate spurious dependencies.

In rare cases, vector optimisation of $\oplus$ produces a wish-list containing an index and/or length function. Though there is scope to optimise upstream code with respect to such wish-lists the current implementation makes no attempt to propagate such wish-lists further upstream.

The translated code for `scan` functions are vector-optimised in an almost identical way to the corresponding reduce-translations.

Summary and Preview This concludes our description of vector optimisation. The vector optimiser, because it analyses data requirements that are dependent on other runtime data, is quite sophisticated. Most of its complexity resides in the rules handling `map`-translations. Next, we describe the, slightly less-complex, task of tuple-optimisation.

5.5 Tuple optimisation

Tuple optimisation is used to prevent the transmission of surplus elements of tuples to downstream code[32]. The strategy employed by the tuple-optimiser is:

Moving data-narrowing operations on upstream.

The data-narrowing operations on tuples are any address function, or composed sequence of address functions such as π_2, π_1, and $\pi_1 \cdot \pi_1$. The process of pushing these

[32]Note that tuple optimisation is not strictly limited to tuples. In the rare case where the input to a function is not a tuple but a single value, and that single value is not needed, tuple-optimisation will eliminate the code that produces that single value.

$$< \textit{filter} > \cdot < \textit{unoptimised code} >$$
$$(a)$$

$$< \textit{optimised code} > \cdot < \textit{filter} > \cdot < \textit{unoptimised code} >$$
$$(b)$$

$$< \textit{optimised code} > \cdot < \textit{filter} >$$
$$(c)$$

Figure 88. A BMF program; at the beginning of tuple-optimisation (part (a)), in the middle of tuple-optimisation (part (b)), and at the end of tuple-optimisation (part (c)).

operations upstream involves the periodic creation and destruction of a specialised type of wish-list called a *filter*.

5.5.1 The filter expression

Figure 88 gives a conceptual view of role of a filter expression during tuple-optimisation.

Part (a) of figure 88 shows a program, prior to tuple-optimisation. At this stage the filter contains an id function to indicate that all of the program's output is required.

Part (b) shows the program in the middle of tuple optimisation. Often, at this stage the filter holds an alltup function containing several addressing functions.

Part (c) shows the program at the end of tuple-optimisation. By this stage the filter will, typically, contain an id function by virtue of the fact that most programs need all of their input values.

The filter expression assumes a very similar role to the wish-list in vector optimisation. It acts as a bridge between optimised and unoptimised code.

Like a wish-list, a filter changes constantly to reflect the needs of optimised downstream code as it moves upstream. Like a wish-list, a filter conveys the data needs of downstream code.

5.5.1.1 What makes the filter different?

The wish-list consists, entirely, of BMF code. A wish-list is just a function. A filter, in most cases, just encapsulates BMF code but, under certain conditions it can encapsulate a non-BMF value called: null.

The role of null Tuple-optimisation eliminates code producing values that are not needed by functions downstream. It is guided, in this task, by the contents of the filter. Any values not referenced in the filter are not needed and any upstream code producing them is removed. In extreme cases, the downstream code needs no specific data. For example, consider the tuple optimisation of:

$$\mathsf{true} \cdot f \cdot g$$

Tuple optimisation starts by placing a filter, containing an id function, to the left of the program:

$$\mathsf{filter}(\mathsf{id}) \cdot \mathsf{true} \cdot f \cdot g$$

Note that this expression is not a valid BMF program, it contains a non-BMF filter expression. Also note that it could, at this point, be easily converted to a valid BMF program by replacing the filter expression with its contents.

The next step of tuple-optimisation produces:

$$\mathsf{true} \cdot \mathsf{filter}(\mathsf{null}) \cdot f \cdot g$$

which says that the constant function true has no real data requirements, it just needs data, of some kind, to trigger its evaluation. The null term is used to denote this *null* data requirement. null is not a function, it cannot accept input or return a value, it is just a term that can appear in a filter expression.

The effect of the null filter, on the upstream code, is profound. Because it doesn't need to produce any particular value, the upstream code doesn't need to take any particular form. Taking advantage of this fact, the tuple-optimiser replaces upstream functions with id functions:

$$\mathsf{true} \cdot \mathsf{id} \cdot \mathsf{id} \cdot \mathsf{filter}(\mathsf{null})$$

Because it contains a filter term, the above is still not a program so, as a last step, the tuple-optimiser converts filter(null) term into an id function:

$$\mathsf{true} \cdot \mathsf{id} \cdot \mathsf{id} \cdot \mathsf{id}$$

After one final run of the *Remove_ids* rule-set, all of the id functions disappear to leave the fully optimised program:

true

Note, that the above is an extreme example of the effect of a null filter. In most program null filters are quickly subsumed by merging with other filters from other parts of the program.

Why is null used only by the tuple-optimiser? The wish-list in the vector-optimiser and the filter-expression in the tuple optimiser are corresponding parts of different processes and, as a result their needs are different. The vector optimiser's wish-list precisely expresses which vector elements are needed but it may also contain functions accessing redundant tuple elements. If the wish-list does contain such redundant access it is not the job of the vector optimiser to do anything about it. Without this obligation, the ordinary functions found in BMF are enough to express any wish-list the vector-optimiser might require.

The tuple optimiser uses the filter expression to precisely convey the values that are needed by downstream code. In most cases, the existing set of BMF functions is enough. However, there is, occasionally, a need to encode the fact that no data, in particular, is required. There is no convenient BMF function to perform this encoding. The null term fulfills this role.

5.5.2 The rules of tuple optimisation

The rules of the tuple optimiser are organised differently to the rules for the vector optimiser. The top-level rules of the vector optimiser capture the various code aggregates produced from Adl source, such as map-translations. Vector optimisation obliterates these aggregates so the tuple optimiser works with individual BMF functions. As a result, tuple-optimisation rules are focused on the elements of BMF syntax rather than on elements of Adl syntax echoed in BMF, as the vector optimiser did. We will structure most of our explanation according to these syntax elements. However, we first have to describe the rule that initiates the tuple-optimisation process.

5.5.2.1 The top level rule

There is one top-level rule for the tuple optimiser. This rule takes an unoptimised program

$$< \textit{unoptimised code} >$$

prepends an identity filter expression to it[33]

$$\mathsf{filter}(\mathsf{id})\cdot < \textit{unoptimised code} >$$

invokes the core rules of tuple optimiser on the combined expression

$$\mathsf{filter}(\mathsf{id})\cdot < \textit{unoptimised code} > \rightarrow < \textit{optimised code} > \cdot\mathsf{filter}(\textit{bexp})$$

and unites the filter with the optimised code to produce the final program:

$$< \textit{optimised code}' >$$

Most of the work of tuple optimisation is done by the core rules. These rules all have conclusions with the following format:

$$\mathsf{filter}(\textit{bexp})\cdot < \textit{unoptimised code} > \rightarrow < \textit{optimised code} > \cdot\mathsf{filter}(\textit{bexp}')$$

That is, the rules at the core of tuple optimisation all attempt to push a filter through unoptimised code in order to create optimised code and a new filter. We describe these rules next.

5.5.2.2 Core tuple-optimisation rules

We start our description with the rules for the simplest constructs and quickly proceed to the rules where the optimiser does most of its work. Our approach is informal; we use rules as a device to help the narrative as opposed to adding formality. We commence our description with the most generic rule. The rule for null filters.

null filters The rule catering for null filters is:

$$\mathsf{filter}(\mathsf{null}) \cdot \textit{bexp} \rightarrow \mathsf{id} \cdot \mathsf{filter}(\mathsf{null})$$

This rule states that any function upstream of a null filter should be converted to id and the null filter should be preserved. Where it is applicable, this rule takes

[33]The tuple-optimiser rules use a comma instead of function composition in the expression filter(id), < *unoptimised code* > because the filter expression is not, strictly speaking, BMF code and as such it cannot be composed with existing BMF code without, first, projecting code from the filter expression. However, it is conceptually useful to think of the filter and the BMF code as being composed so we maintain this minor fiction.

precedence over all other rules, that is, when another rule is also applicable, the above rule is applied in preference.

We have already seen the effect of the null-filter rule in our earlier explanation of the null term. Note that this rule is rarely applied. null filter expressions are common but they are almost always merged with other filter expressions before they get the chance to be applied to the rule above. We will see an example of this merging process in our discussion of alltup optimisation on page 149. Next, we examine the rule that generates null-filters.

Constants The rule for constant functions is:

$$\mathsf{filter}(\mathit{bexp}) \cdot \mathsf{K} \rightarrow \mathsf{K} \cdot \mathsf{filter}(\mathsf{null})$$

That is, any constant function will generate a null-filter. If *bexp* is not null the constant function K is preserved.

Addresses Address functions are the only functions that narrow tuples. The rule for address functions is:

$$\mathsf{filter}(\mathit{bexp}) \cdot^{m} \pi_n \rightarrow \mathsf{id} \cdot \mathsf{filter}(\mathit{bexp} \cdot^{m} \pi_n)$$

The rule captures the address function in the filter expression so it can be propagated upstream until it reaches the corresponding tuple generating alltup function.

There is a strong correspondence between the role played by address functions in tuple-optimisation and the role played by index and length functions in vector optimisation.

Function composition The tuple optimiser has an explicit rule for sequences of composed functions:

$$\frac{\begin{array}{c}\mathsf{filter}(\mathit{bexp}_1) \cdot \mathit{bexp}_2 \rightarrow \mathit{bexp}_2' \cdot \mathsf{filter}(\mathit{bexp}_1') \\ \mathsf{filter}(\mathit{bexp}_1') \cdot \mathit{bexp}_3 \rightarrow \mathit{bexp}_3' \cdot \mathsf{filter}(\mathit{bexp}_1'')\end{array}}{\mathsf{filter}(\mathit{bexp}_1) \cdot \mathit{bexp}_2 \cdot \mathit{bexp}_3 \rightarrow \mathit{bexp}_2' \cdot \mathit{bexp}_3' \cdot \mathsf{filter}(\mathit{bexp}_1'')}$$

The rule works by tuple-optimising the first function to create a result function bexp_2' and a new filter $\mathsf{filter}(\mathit{bexp}_1')$ which is then used as part of the tuple-optimisation of the second function.

The function composition rule is the mechanism for the propagation of filter expressions through the program. It also provides, when used in conjunction with the rule for address functions, a mechanism for refining the focus of address functions appearing in the filter expression. For example, tuple optimisation of the function $\pi_1 \cdot \pi_1 \cdot \pi_1$ proceeds:

$$\begin{array}{ll}
\mathsf{filter}(\mathsf{id}) \cdot \pi_1 \cdot \pi_1 \cdot \pi_1 & \rightarrow \\
\mathsf{id} \cdot \mathsf{filter}(\pi_1) \cdot \pi_1 \cdot \pi_1 & \rightarrow \\
\mathsf{id} \cdot \mathsf{id} \cdot \mathsf{filter}(\pi_1 \cdot \pi_1) \cdot \pi_1 & \rightarrow \\
\mathsf{id} \cdot \mathsf{id} \cdot \mathsf{id} \cdot \mathsf{filter}(\pi_1 \cdot \pi_1 \cdot \pi_1) &
\end{array}$$

At each step above, the filter expression is refined to be more specific about the data it refers to.

Alltup alltup functions are tuple-constructors. They are responsible for the creation of most surplus tuple values in unoptimised code. Not surprisingly then, alltup functions are, typically, the functions most strongly affected by tuple-optimisation.

The rule for tuple optimisation of alltup functions is:

$$\frac{\begin{array}{c} \mathit{Select\_items}(\mathit{bexp}_1 \cdot (\mathit{bexp}_{(2,1)}, \ldots \mathit{bexp}_{(2,n)})^\circ \Rightarrow \mathit{bexp}'_2) \\ \mathit{Compact\_addr\_space}(\mathit{bexp}'_2 \Rightarrow \mathit{bexp}''_2 \cdot \mathsf{filter}(\mathit{bexp}'_1) \end{array}}{\mathsf{filter}(\mathit{bexp}_1) \cdot (\mathit{bexp}_{(2,1)}, \ldots \mathit{bexp}_{(2,n)})^\circ \rightarrow \mathit{bexp}''_2 \cdot \mathsf{filter}(\mathit{bexp}'_1)}$$

The rule has two premises. The first premise calls the *Select_items* rule-set. *Select_items* merges the filter with the alltup function to produce a single result function. Figure 89 illustrates how this merging process takes place. After the merging process is applied, the result function:

$$(\pi_2, + \cdot (\pi_2 \cdot \pi_1, \pi_2)^\circ)^\circ$$

produces only the data required by downstream code. This is all well-and-good. However, close scrutiny of the function above reveals that the set of address functions it contains:

$$\{\pi_2 \cdot \pi_1, \pi_2\}$$

does not *cover* the whole of input address space. In particular, there is no value referenced by the function $\pi_1 \cdot \pi_1$.

We eliminate this superfluous input and, simultaneously, create a new filter term by rewriting $(\pi_2, + \cdot (\pi_2 \cdot \pi_1, \pi_2)^\circ)^\circ$ to what it would be if it received *only* the input

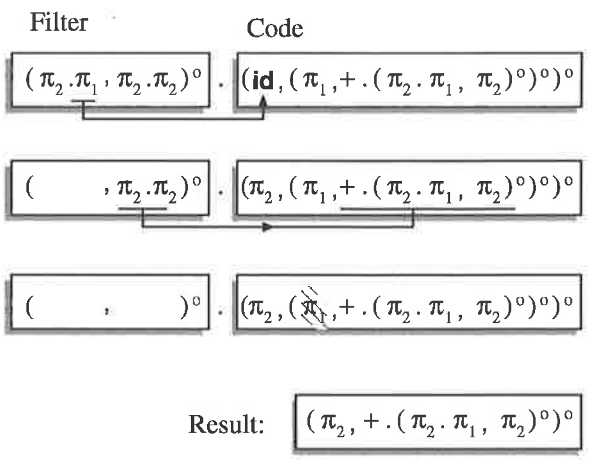

Filter
Code
$(\pi_2.\pi_1, \pi_2.\pi_2)^\circ$
$(\mathbf{id}, (\pi_1, +.(\pi_2.\pi_1, \pi_2)^\circ)^\circ)^\circ$
$(\ , \pi_2.\pi_2)^\circ$
$(\pi_2, (\pi_1, +.(\pi_2.\pi_1, \pi_2)^\circ)^\circ)^\circ$
$(\ , \ )^\circ$
$(\pi_2, (\pi_1, +.(\pi_2.\pi_1, \pi_2)^\circ)^\circ)^\circ$
Result:
$(\pi_2, +.(\pi_2.\pi_1, \pi_2)^\circ)^\circ$

Figure 89. An illustration of how *Select_items* merges a filter expression and an alltup function to produce a result function.

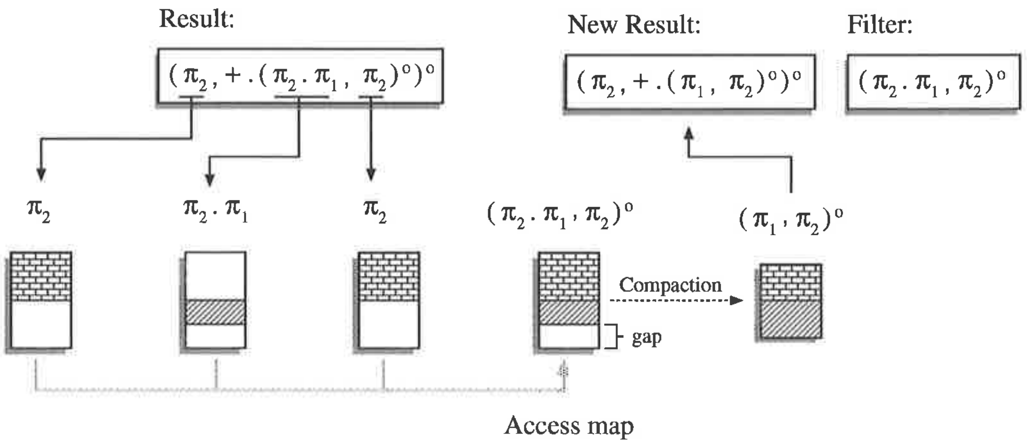

Result:
$(\pi_2, +.(\pi_2.\pi_1, \pi_2)^\circ)^\circ$
New Result:
$(\pi_2, +.(\pi_1, \pi_2)^\circ)^\circ$
Filter:
$(\pi_2.\pi_1, \pi_2)^\circ$
π_2
$\pi_2.\pi_1$
π_2
$(\pi_2.\pi_1, \pi_2)^\circ$
$(\pi_1, \pi_2)^\circ$
Compaction
gap
Access map

Figure 90. The action of *Compact_addr_space* on the result of *Select_items*.

it needed and, in addition, generating a new filter term that selects only this input. In the tuple optimiser, the *Compact_addr_space* rule-set performs this task of result-modification and filter-generation.

Figure 90 shows the effect of *Compact_addr_space* on the result function from figure 89. The left-hand-side of the figure shows that the address functions in the alltup function are used to form an access map of the input value. If the input value is a tuple and not all values of the input tuple are accessed, there will be gaps, indicated in white, in the access map. The right-hand-side of the figure shows that the result function is modified so there are no gaps in its access map and a new filter expression is built to specify the data required.

map The rule for tuple optimisation of map functions is:

$$\frac{\mathsf{filter}(bexp_1) \cdot bexp_2 \rightarrow bexp_2' \cdot \mathsf{filter}(bexp_1')}{\mathsf{filter}((bexp_1)*) \cdot (bexp_2)* \rightarrow (bexp_2') * \cdot \mathsf{filter}((bexp_1')*)}$$

The premise of the rule processes the code inside the map function with respect to the code inside the map function in the filter. The resulting code and filter are re-injected back into map function in the rule's conclusion. The appearance of a map function in a filter indicates that the expected output of the upstream function is a vector[34]

Allvec allvec functions are morphologically similar to alltup functions. The tuple-optimiser rule propagates the filter into each of the functions contained in the allvec function. The individual filters produced by this process are then united to form a new filter using the *Compact_addr_space* rule-set used in the rule for alltup functions.

Distl Even after vector optimisation, there may still be redundant instances of distl embedded in the code. The tuple-optimiser can safely eliminate a distl function if the filter expression contains only addresses prefixed with π_2. In this case, only elements of distl's second parameter are accessed by downstream code and the distl function can be safely eliminated. The code from the old expression, stripped of its leading π_2 functions replaces the distl function. A new filter expression is generated containing just π_2 and positioned just upstream of the new code replacing distl.

If the filter expression contains references other than π_2 the distl function is left intact and the new filter expression contains id.

zip A zip function is redundant if the filter exclusively references one side of each pair of the vector produced by that zip. If, for example, the filter expression contains only addresses starting with π_1 then the second half of each pair isn't needed and the zip function can be replaced by the filter expression stripped of its leading π_1 functions. A new filter expression is generated containing just π_1.

The above process is mirrored if the filter expression contains only addresses starting with π_2.

[34]The converse - a vector emanating from an upstream function implies a filter containing a map function - is not true. A filter downstream of a map can also be an id function. Extra rules are required to cope with this special case in various parts of the tuple-optimiser. Later versions of the tuple optimiser are likely to avoid these special cases by more directly exploiting the type information accompanying the code.

If the addresses in the filter expression contains references to both π_1 and π_2 or contains a leading id then the zip function is kept intact and a new id filter is created.

If The data requirements of an if function is the union of the data requirements of each of its parts. The data requirements of the consequent and alternative parts of the if functions are determined by optimising each of them with respect to the current filter expression. In contrast, the data needed by the predicate is determined by optimising it with respect to an id filter. The id filter is used for the predicate because we can safely assume that all of the output of the predicate function, a boolean value, is required for the program to work properly.

While The current implementation does not tuple-optimise while functions with respect to the surrounding code. Instead, the inside of while functions are tuple-optimised with-respect to an id filter. The data needed by the while function is determined by a combination of the needs of the iterative function and the needs of the predicate. The iterative function, in particular, can consume its own input so it must be optimised with respect to its own input requirements. Such optimisation is future work.

All other functions All functions, other than those already mentioned are captured by the default rule:

$$\mathsf{filter}(bexp_1) \cdot bexp_2 \rightarrow bexp_1 \cdot bexp_2 \cdot \mathsf{filter}(\mathsf{id})$$

This rule, like the rule for while inserts the code in the filter downstream of the function and produces a new id filter expression. This catch-all rule attempts no real optimisation.

Note, that even though this rule applies to all variations of reduce and scan the tuple optimiser is invoked upon the code embedded in these functions during vector optimisation. That is, by the time this rule applies to reduce and scan functions the code in inside these functions has already been tuple-optimised.

Summary This concludes our description of the rules of the tuple optimiser. We have outlined the top-level rules and canvassed the core issues that arise during tuple optimisation. Note, that as with vector optimisation, we have omitted many details, hidden in auxiliary rule sets, whose description is beyond the scope of this report.

Next we summarise the findings arising directly from our work with the vector and tuple optimisers. Starting with a review of the performance of the code produced by the combined optimisers.

5.6 Findings

Our findings for this chapter are divided into

- findings relating to the performance of the optimiser and
- lessons learnt during the construction of our implementation.

Both aspects provide useful information. The first helps to confirm the effectiveness of our approach. The second describes important features of our approach.

We describe these aspects in turn.

5.6.1 Performance

There are many ways to view the performance of an implementation but the primary concern is the efficiency of the target code. Secondary measures of performance include, the reliability of the optimisation process, the time it takes the optimisation process to run, and the size and elegance of the code produced by the optimiser. We examine efficiency of target code first.

5.6.1.1 Efficiency of target code

At the end of the last chapter, we saw that the translation process, though reliable, often produced inefficient code. Now, at the end of this chapter, we argue that the optimisation process we have just described greatly enhances the efficiency of that code. In the following, we extend the examples from the previous chapter to compare the performance of optimiser code to that of both translator code and idealised code from the last chapter. We also include new examples for other representative constructs. The examples we present here are:

`map_map_addconst.Adl` a program containing a very simple nested `map` function.

`sum.Adl` a very simple `reduce`.

```
main a:vof vof int :=
   let
      f x :=
         let
            g y := y + 2
         in
            map (g,x)
         endlet
   in
      map (f,a)
   endlet
```

Figure 91. `map_map_addconst.Adl`, an Adl program that adds a constant to each element of a nested vector

`mss.Adl` two `map`'s and a `scan` function with a detailed binary-operator.

`finite_diff.Adl` a `map` containing multiple vector indexing operations.

`transpose.Adl` a nested `map` using indexing operations to implement transposition of nested vectors.

Some of these examples contain code with many details. A deep understanding of the details is not required in order to read this section. The text points out the salient properties of the code in each case. Details are included for the sake of completeness.

The reader should note that our examination of the impact of the optimiser extends beyond this chapter. Further examples, at the end of the next chapter, serve to demonstrate that the optimiser ultimately improves the performance of parallelised code.

Adding a constant to each element of a nested vector Figure 91 shows the source code for `map_map_addconst.Adl`, a program we examined at the end of the last chapter. Figure 92 shows the translator code, efficient hand-crafted code and optimiser code, respectively, for `map_map_addconst.Adl`. The hand-crafted and optimiser code are identical. The optimiser is often able to produce a good match to hand-crafted code when presented with simple programs.

$$((+ \cdot (\pi_2, 2)^\circ) * \cdot \mathsf{distl} \cdot (\mathsf{id}, \pi_2)^\circ \cdot \mathsf{id}) * \cdot \mathsf{distl} \cdot (\mathsf{id}, \mathsf{id})^\circ \cdot \mathsf{id}$$

(a)

$$((+ \cdot (\mathsf{id}, 2)^\circ)*)*$$

(b)

$$((+ \cdot (\mathsf{id}, 2)^\circ)*)*$$

(c)

Figure 92. Translator code (part (a)), hand-crafted code (part (b)), and optimiser code (part (c)) for `map_map_addconst.Adl`

Figure 93 shows the relative performance of all three versions when applied to the data:

$$[[1, 2, 3], [4, 5, 6], [7, 8, 9]]$$

The curves for the hand-crafted and optimiser code are coincident.

Summing a vector of numbers Figure 94 shows the source code for `sum.Adl` a program, introduced in the last chapter, that uses `reduce` to perform a summation over a one-dimensional vector. Figure 95 shows the translator code, hand-crafted code and the optimiser code, respectively, for `sum.Adl` As can be seen, the hand-crafted and fully-optimised code are the same and their performance, as shown in figure 96 is much better than that of the raw translator code. The reason for the large difference is primarily due to the vector optimiser rules for reduce that recognise that the binary operator +, in this case, makes no references to values other than its immediate operands. The optimised code is, essentially, the translator code with all of the paraphernalia used to transport references to non-operands removed.

Partial Maximum-Segment-Sums An interesting operation that can be elegantly expressed using the primitives provided by Adl is *maximum-segment-sum*.

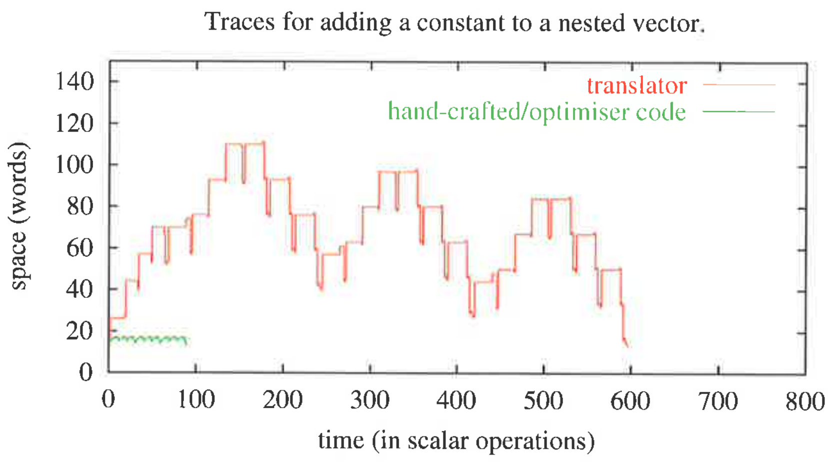


Figure 93. Time/Space graph of sequential evaluation of translator, hand-crafted and optimiser code for `map_map_addconst.Adl` applied to the nested vector $[[1, 2, 3], [4, 5, 6], [7, 8, 9]]$. The curves for the hand-crafted and optimiser code are coincident.

```
main a: vof int :=
let
   add(x,y) := x + y
in
  reduce (add,0,a)
endlet
```

Figure 94. `sum.Adl`, a program to sum a vector of integers

$$\mathsf{if}(\neq \cdot (0, \# \cdot \pi_2)^\circ, \pi_2 \cdot ((\pi_1 \cdot \pi_1, + \cdot (\pi_1 \cdot \pi_1, (\pi_2 \cdot \pi_1, \pi_2 \cdot \pi_2)^\circ)^\circ)^\circ)/ \cdot \mathsf{distl}, 0 \cdot \pi_1) \cdot (\mathsf{id}, \mathsf{id})^\circ$$

(a)

$$+/_0$$

(b)

$$+/_0$$

(c)

Figure 95. Translator code (a), hand-crafted code (b) and optimiser-code (c) for `sum.Adl`

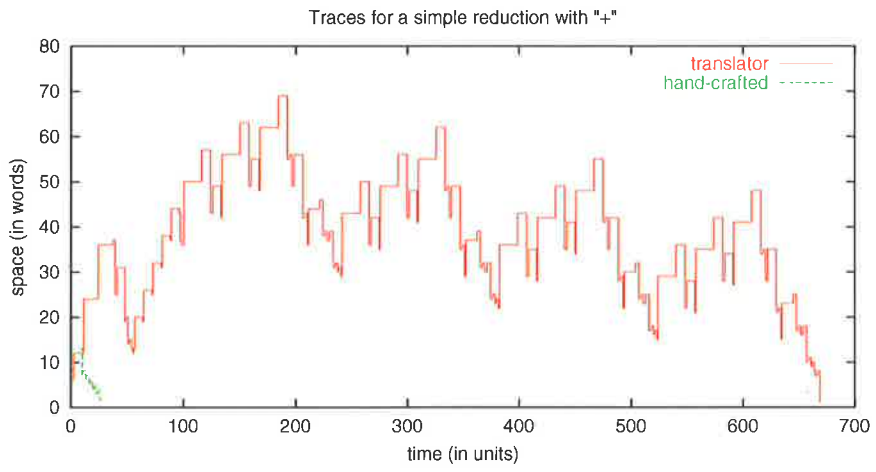


Figure 96. Time/Space graph of sequential evaluation of translator, hand-crafted and optimiser code for `reduce_plus.Adl` applied to the vector $[1, 2, 3, 4, 5]$. The curves for the hand-crafted and optimiser code are coincident.

This operation calculates the maximum sum of all the contiguous segments of a vector. So, for example, the maximum-segment-sum of the vector:

$$[1, 2, -7, 8, -1, 4, 1, -3, 2, 3]$$

is 14 which is the sum of the segment $[8, -1, 4, 1, -3, 3, 2]$. A parallel solution to the maximum-segment-sum problem, framed in terms of map and reduce, is presented by Murray Cole in [36]. Since we have already examined the optimiser's treatment of reduce we present a variation of maximum-segment-sum, based on scan that calculates the maximum-segment-sum for all of the initial segments of a vector. The output of this operation, that we shall name *partial-maximum-segment-sums*, when applied to the vector $[1, 2, -7, 8, -1, 4, 1, -3, 2, 3]$ is the vector:

$$[1, 3, 3, 8, 8, 11, 12, 12, 12, 14]$$

Figure 97 shows Adl source code for partial-maximum-segment-sums. The basic structure of the program is:

- a map to inject each value of the original vector into a 4-tuple.
- A scan to apply a binary operator `oplus` to the elements of the vector to produce a vector of 4-tuples with a result value embedded in each one and
- a map applying the function `first` to each tuple to project out each element of the result vector.

```
main a: vof int :=
  let
    plus (x,y) := x + y;
    max (x,y) := if (x > y) then x else y endif;
    oplus ((mssx,misx,mcsx,tsx),(mssy,misy,mcsy,tsy)) :=
      (max(max(mssx,mssy),plus(mcsx,misy)),
       max(misx,plus(tsx,misy)),
       max(plus(mcsx,tsy),mcsy),tsx+tsy);
    f x :=
       let
         mx0 := max(x,0)
       in
         (mx0,mx0,mx0,x)
       endlet;
    first (mss,mis,mcs,ts) := mss
  in
    map(first,scan(oplus,map(f,a)))
  endlet
```

Figure 97. Source code for `mss.Adl`, a program that calculates partial-maximum-segment-sums over an input vector. At the core of this program is a `scan` function that contains the binary operation `oplus` that performs all of the book keeping required to keep track of the current maximum segment sum.

The `oplus` operation is central to how maximum-segment-sum works. We leave a detailed explanation of `oplus` to Cole's article. For our purposes, it is enough to know that partial-maximum-segment-sums provides an interesting working example that uses both `map` and `scan` functions in a non-trivial context.

Figure 98 shows the translator, hand-crafted and optimised BMF code for `mss.Adl` The translator code in (part (a)) contains all of the machinery required to transport any value in scope to each application of the binary operator. The hand-crafted code (part (b)) and optimised code (part (c)) are streamlined to take advantage of the fact that the binary operator accesses only its operands. Figure 99 shows the relative performance of the three BMF versions of `mss.Adl`. Both the hand-crafted and optimised code exhibit much better performance than the translator code, primarily because they avoid transporting all values in scope to every instance of the binary function. The optimised code is slightly less efficient than the hand-crafted code. This appears to be due to extra code in the optimised version that flattens the nested

$$\begin{array}{l}
({}^4\pi_1 \cdot \pi_2) * \cdot \mathsf{distl} \cdot \\
(\mathsf{id}, \mathsf{if}(\neq \cdot (0, \# \cdot \pi_2)^\circ, \\
(\pi_2) * \cdot (\pi_1 \cdot \pi_1, \\
\qquad (\mathsf{if}(> \cdot (\pi_1 \cdot \pi_2, \pi_2 \cdot \pi_2)^\circ, \pi_1 \cdot \pi_2, \pi_2 \cdot \pi_2) \cdot \\
\qquad\qquad (\mathsf{id}, (\mathsf{if}(> \cdot (\pi_1 \cdot \pi_2, \pi_2 \cdot \pi_2)^\circ, \pi_1 \cdot \pi_2, \pi_2 \cdot \pi_2) \cdot \\
\qquad\qquad (\mathsf{id}, ({}^4\pi_1 \cdot \pi_1 \cdot \pi_2, {}^4\pi_1 \cdot \pi_2 \cdot \pi_2)^\circ)^\circ, \\
\qquad + \cdot (\pi_1 \cdot \pi_2, \pi_2 \cdot \pi_2)^\circ \cdot (\mathsf{id}, ({}^4\pi_3 \cdot \pi_1 \cdot \pi_2, {}^4\pi_2 \cdot \pi_2 \cdot \pi_2)^\circ)^\circ)^\circ)^\circ, \\
\qquad \mathsf{if}(> \cdot (\pi_1 \cdot \pi_2, \pi_2 \cdot \pi_2)^\circ, \pi_1 \cdot \pi_2, \pi_2 \cdot \pi_2) \cdot \\
\qquad\qquad (\mathsf{id}, ({}^4\pi_2 \cdot \pi_1 \cdot \pi_2, + \cdot (\pi_1 \cdot \pi_2, \pi_2 \cdot \pi_2)^\circ \cdot (\mathsf{id}, ({}^4\pi_4 \cdot \pi_1 \cdot \pi_2, {}^4\pi_2 \cdot \pi_2 \cdot \pi_2)^\circ)^\circ)^\circ)^\circ, \\
\qquad \mathsf{if}(> \cdot (\pi_1 \cdot \pi_2, \pi_2 \cdot \pi_2)^\circ, \pi_1 \cdot \pi_2, \pi_2 \cdot \pi_2) \cdot \\
\qquad\qquad (\mathsf{id}, (+ \cdot (\pi_1 \cdot \pi_2, \pi_2 \cdot \pi_2)^\circ \cdot (\mathsf{id}, ({}^4\pi_3 \cdot \pi_1 \cdot \pi_2, {}^4\pi_4 \cdot \pi_2 \cdot \pi_2)^\circ)^\circ, {}^4\pi_3 \cdot \pi_2 \cdot \pi_2)^\circ)^\circ, \\
\qquad + \cdot ({}^4\pi_4 \cdot \pi_1 \cdot \pi_2, {}^4\pi_4 \cdot \pi_2 \cdot \pi_2)^\circ)^\circ \\
\cdot (\pi_1 \cdot \pi_1, (\pi_2 \cdot \pi_1, \pi_2 \cdot \pi_2)^\circ)^\circ)^\circ /\!\!/ \cdot \mathsf{distl}, \\
(0, 0, 0, 0)^\circ \cdot \pi_1) \cdot \\
(\mathsf{id}, ((\pi_2, \pi_2, \pi_2, \pi_2 \cdot \pi_1)^\circ \cdot \mathsf{id} \cdot \\
(\mathsf{id}, \mathsf{if}(> \cdot (\pi_1 \cdot \pi_2, \pi_2 \cdot \pi_2)^\circ, \pi_1 \cdot \pi_2, \pi_2 \cdot \pi_2) \cdot (\mathsf{id}, (\pi_2, 0)^\circ)^\circ)^\circ) * \cdot \\
\mathsf{distl} \cdot (\mathsf{id}, \mathsf{id})^\circ)^\circ)^\circ \cdot \mathsf{id}
\end{array}$$

(a)

$$\begin{array}{l}
({}^4\pi_1) * \cdot \\
(\ \ \mathsf{if}(> \cdot (\pi_1, \pi_2)^\circ, \pi_1, \pi_2) \cdot \\
\qquad (\mathsf{if}(> \cdot ({}^4\pi_1 \cdot \pi_1, {}^4\pi_1 \cdot \pi_2)^\circ, {}^4\pi_1 \cdot \pi_1, {}^4\pi_1 \cdot \pi_2), + \cdot ({}^4\pi_3 \cdot \pi_1, {}^4\pi_2 \cdot \pi_2)^\circ)^\circ \\
\quad \mathsf{if}(> \cdot (\pi_1, \pi_2)^\circ, \pi_1, \pi_2) \cdot {}^4\pi_2 \cdot \pi_1, + \cdot ({}^4\pi_4 \cdot \pi_1, {}^4\pi_2 \cdot \pi_2)^\circ)^\circ \\
\quad \mathsf{if}(> \cdot (\pi_1, \pi_2)^\circ, \pi_1, \pi_2) \cdot (+ \cdot ({}^4\pi_3 \cdot \pi_1, \pi_4 \cdot \pi_2)^\circ, {}^4\pi_3 \cdot \pi_2)^\circ, \\
\quad + \cdot ({}^4\pi_4 \cdot \pi_1, \pi_4 \cdot \pi_2)^\circ)^\circ /\!\!/ \cdot \\
((\pi_1, \pi_1, \pi_1, \pi_2)^\circ \cdot (\mathsf{if}(> \cdot (\mathsf{id}, 0)^\circ, \mathsf{id}, 0), \mathsf{id})^\circ) *
\end{array}$$

(b)

$$\begin{array}{l}
({}^4\pi_1) * \cdot \\
(\ \ \mathsf{if}(> \cdot (\mathsf{if}(> \cdot ({}^8\pi_1, {}^8\pi_2)^\circ, {}^8\pi_1, {}^8\pi_2), + \cdot ({}^8\pi_3, {}^8\pi_4)^\circ)^\circ, \\
\qquad \mathsf{if}(> \cdot ({}^8\pi_1, {}^8\pi_2)^\circ, {}^8\pi_1, {}^8\pi_2), + \cdot ({}^8\pi_3, {}^8\pi_4)^\circ), \\
\quad \mathsf{if}(> \cdot ({}^8\pi_5, + \cdot ({}^8\pi_6, {}^8\pi_4)^\circ)^\circ, {}^8\pi_5, + \cdot ({}^8\pi_6, {}^8\pi_4)^\circ), \\
\quad \mathsf{if}(> \cdot (+ \cdot ({}^8\pi_3, {}^8\pi_7)^\circ, {}^8\pi_8)^\circ, \\
\quad + \cdot ({}^8\pi_3, {}^8\pi_7)^\circ, {}^8\pi_8), \\
\quad + \cdot ({}^8\pi_6, {}^8\pi_7)^\circ)^\circ \cdot \\
({}^4\pi_1 \cdot \pi_1, {}^4\pi_1 \cdot \pi_2, {}^4\pi_3 \cdot \pi_1, {}^4\pi_2 \cdot \pi_2, {}^4\pi_2 \cdot \pi_1, {}^4\pi_4 \cdot \pi_1, {}^4\pi_4 \cdot \pi_2, {}^4\pi_3 \cdot \pi_2)^\circ /\!\!/ \cdot \\
((\pi_1, \pi_1, \pi_1, \pi_2)^\circ \cdot (\mathsf{if}(> \cdot (\mathsf{id}, 0)^\circ, \mathsf{id}, 0), \mathsf{id})^\circ) *
\end{array}$$

(c)

Figure 98. BMF code for `mss.Adl`. Raw translator code (part(a)). Hand-crafted-code (part (b)). Fully optimised code (part (c)). The superscript before some π functions indicates the arity of the tuple being projected from (e.g. ${}^4\pi_2$ accesses the second element of a 4-tuple.)

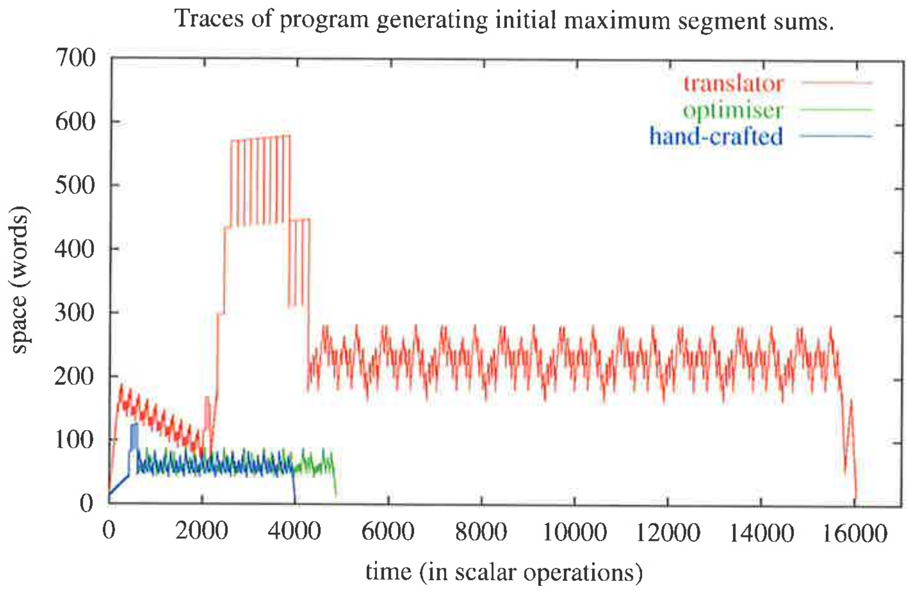


Figure 99. Time/Space graph of sequential evaluation of translator, hand-crafted and optimiser code for `mss.Adl` applied to the vector $[1, 2, -7, 8, -1, 4, 1, -3, 2, 3]$.

input tuple to the binary function into an 8-tuple. Such flattening makes downstream access to tuple elements more efficient[35] but incurs a cost itself. In this case the cost of flattening outweighs the benefits but the overall effect is marginal.

Finite difference This example was featured at the end of chapter 4 and helps to illustrate the impact of optimisation of vector references inside map functions. The source code for `finite_diff.Adl` is shown in figure 100. As can be seen from the code, `finite_diff.Adl` consists of a map whose function parameter contains three vector references. Figure 101 shows four different BMF renditions of `finite_diff.Adl`. The code in parts (a), (b) and (c) is directly transcribed from the end of chapter 4. Parts (a) and (b) are, respectively, the translator code and hand-crafted code, both, without select. Both of these versions distribute a copy of the entire input vector to each invocation of the map body. Part (c) is hand-crafted code using select to distribute only the required elements to each invocation of the map body. Part (d) is new code, produced by the optimiser from the translator code in part (a). This

[35]Access to an element of a pair of 4-tuples is a two-step process, specifically, selecting the correct 4-tuple and then selecting the correct element of the chosen 4-tuple. Access to an element of an un-nested 8-tuple is a one-step process and is thus more efficient.

```
main a: vof int :=
  let
    stencil x := a!x + a!(x-1) + a!(x+1);
    addone x := x + 1;
    element_index := map(addone,iota ((# a)-2))
  in
    map (stencil, element_index)
  endlet
```

Figure 100. `finite_diff.Adl` an Adl program to perform a simple finite-difference computation.

$$(+ \cdot (+ \cdot (! \cdot (\pi_1 \cdot \pi_1, \pi_2)^\circ, ! \cdot (\pi_1 \cdot \pi_1, - \cdot (\pi_2, 1)^\circ)^\circ)^\circ, ! \cdot (\pi_1 \cdot \pi_1, + \cdot (\pi_2, 1)^\circ)^\circ)^\circ) * \cdot$$
$$\mathsf{distl} \cdot (\mathsf{id}, \pi_2)^\circ \cdot \mathsf{id} \cdot$$
$$(\mathsf{id}, (+ \cdot (\pi_2, 1)^\circ) * \mathsf{distl} \cdot (\mathsf{id}, \mathsf{iota} \cdot - \cdot (\# \cdot \mathsf{id}, 2)^\circ)^\circ)^\circ$$

(a)

$$(+ \cdot (+ \cdot (! \cdot (\pi_1, \pi_2)^\circ, ! \cdot (\pi_1, - \cdot (\pi_2, 1)^\circ)^\circ)^\circ, ! \cdot (\pi_1, + \cdot (\pi_2, 1)^\circ)^\circ)^\circ) * \cdot$$
$$\mathsf{distl} \cdot$$
$$(\mathsf{id}, (+ \cdot (\mathsf{id}, 1)^\circ) * \cdot \mathsf{iota} \cdot - \cdot (\# \cdot \mathsf{id}, 2)^\circ)^\circ$$

(b)

$$(+ \cdot (\pi_1 \cdot \pi_1, + \cdot (\pi_2 \cdot \pi_1, \pi_2)^\circ)^\circ) * \cdot$$
$$\mathsf{zip} \cdot (\mathsf{zip} \cdot (\pi_1 \cdot \pi_1, \pi_2 \cdot \pi_1)^\circ, \pi_2)^\circ \cdot$$
$$((\mathsf{select}, \mathsf{select} \cdot (\pi_1, (+ \cdot (\mathsf{id}, 1)^\circ) * \cdot \pi_2)^\circ)^\circ, \mathsf{select} \cdot (\pi_1, (- \cdot (\mathsf{id}, 1)^\circ) * \cdot \pi_2)^\circ)^\circ \cdot$$
$$(\mathsf{id}, (+ \cdot (\mathsf{id}, 1)^\circ) * \cdot \mathsf{iota} \cdot - \cdot (\#, 2)^\circ)^\circ$$

(c)

$$(+ \cdot (+ \cdot \pi_1, \pi_2)^\circ) * \cdot \mathsf{zip} \cdot$$
$$(\mathsf{zip} \cdot (\mathsf{select}, \mathsf{select} \cdot (\pi_1, (-) * \cdot \mathsf{zip} \cdot (\mathsf{id}, \mathsf{repeat} \cdot (1, \#)^\circ)^\circ \cdot \pi_2)^\circ)^\circ,$$
$$\mathsf{select} \cdot (\pi_1, (+) * \cdot \mathsf{zip} \cdot (\mathsf{id}, \mathsf{repeat} \cdot (1, \#)^\circ)^\circ \cdot \pi_2)^\circ)^\circ \cdot$$
$$(\mathsf{id}, (+ \cdot (\mathsf{id}, 1)^\circ) * \cdot \mathsf{iota} \cdot - \cdot (\#, 2)^\circ)^\circ$$

(d)

Figure 101. Four versions of BMF code for `finite_diff.Adl`. Raw translator code (part (a)). Hand-crafted code without select (part (b)). Hand-crafted code with select (part (c)). Fully optimised code (part(d)).

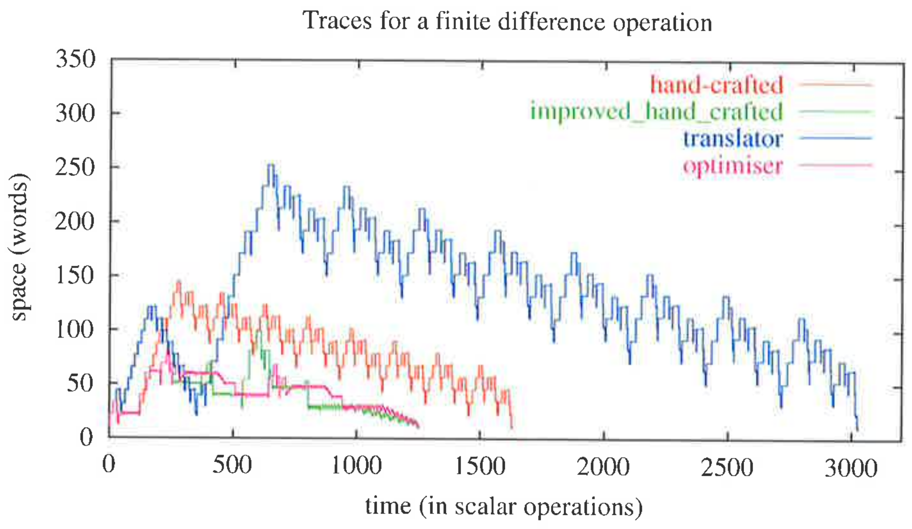


Figure 102. The relative performance of translator-code, hand-crafted code without select, hand-crafted code with select and optimised code when applied to the input vector $[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$. Note the near-identical performance of the best hand-crafted code and the optimiser code.

code, like part (c), uses select to transport just the required vector elements to each invocation.

The relative performance of each of these versions, when applied to a small input vector is shown in figure 102. The performances of the translator code and both versions of hand-crafted code are the same as they were at the end of chapter 4. The performance of the optimised code, in part (d), is almost identical to the corresponding hand-crafted code in part (c). This is undoubtedly a good result for the optimiser.

Transpose Figure 103 shows Adl source code for a a program that transposes a nested input vector. The code consists of a nested map where the most deeply embedded function contains a nested vector-indexing expression, with indices reversed, to produce a transposition of the input vector[36].

Figure 104 shows the translator code, hand-crafted code and optimised code for `transpose.Adl`. The translator code relies on distributing a copy of the nested vector to every invocation of the inner map's body. The hand-crafted code is simply a transpose operation on the two dimensions of the input vector. The optimiser code is more detailed than the hand-crafted code. It contains a transpose function that

[36]Note that the program will only produce a correct transposition if the longest inner-vector is `a!0`. If this is not the case then some truncation of rows will occur.

```
main a:vof vof int :=
   let
      f x := let
                g y := a!y!x
             in
                map (g,iota (# a))
             endlet
   in
      map(f,iota (# (a!0)))
   endlet
?
```

Figure 103. `transpose.Adl`, a program to transpose a nested input vector.

$$((! \cdot (! \cdot (\pi_1 \cdot \pi_1, \pi_2)^\circ, \pi_2 \cdot \pi_1)^\circ) * \cdot \mathsf{distl} \cdot (\mathsf{id}, \mathsf{iota} \cdot \# \cdot \pi_1)^\circ \cdot \mathsf{id}) * \cdot$$
$$\mathsf{distl} \cdot (\mathsf{id}, \mathsf{iota} \cdot \# \cdot ! \cdot (\mathsf{id}, 0)^\circ)^\circ \cdot \mathsf{id}$$

(*a*)

$$\mathsf{transpose}_{(0,1)}$$

(*b*)

$$(\mathsf{select} \cdot (\pi_1, \mathsf{iota} \cdot \pi_2)^\circ) * \cdot$$
$$\mathsf{zip} \cdot (\mathsf{select} \cdot (\mathsf{transpose}_{(1,0)} \cdot {}^3\pi_1, {}^3\pi_2)^\circ, \mathsf{repeat} \cdot ({}^3\pi_3, \# \cdot {}^3\pi_2)^\circ)^\circ \cdot$$
$$(\pi_1, \mathsf{iota} \cdot \# \cdot ! \cdot (\pi_1, 0)^\circ, \pi_2 \cdot \pi_2)^\circ \cdot (\mathsf{id}, (\mathsf{id}, \#)^\circ)^\circ$$

(*c*)

Figure 104. Three BMF versions of `transpose.Adl`. The translator-code, shown in part (a), the hand-crafted program, shown in part (b), and the optimised program, shown in part (c).

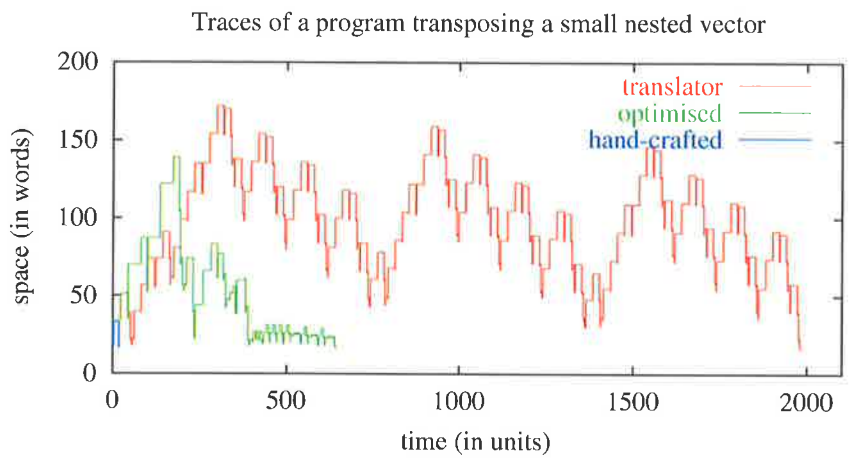


Figure 105. The relative performance of translator-code, hand-crafted code and optimised BMF code for `transpose.Adl` when applied to the nested input vector $[[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]$.

swaps the appropriate dimensions but it also contains code that performs a nested select on the transposed vector. It turns out that this nested select is semantically equivalent to a id but the current implementation does not have the capacity to detect this.

The relative efficiency of the translator code, the hand-crafted code, and the optimiser code for `transpose.Adl` is shown in figure 105. The performance of the optimiser code is a substantial improvement on the performance of the translator code but there remains a significant gap between the performance of the optimiser code and that of the hand-crafted code. The good news is that this performance gap does not grow quickly with the size of the input. Figure 106 shows the performance of the same programs applied to a 4×6 vector (twice as many elements as the previous vector).

As can be seen, the performance gap between translator code and optimised code has grown substantially more than the performance gap between the optimised code and hand-crafted code. Optimiser code appears slower than the hand-crafted code by only a constant factor which augers well for future implementations.

Summary of the performance of target code The examples we have just presented are a sample of the experiments that we have run. The behaviour of this sample is representative of the behaviour of target code in other code we have tested. In all tested programs, the optimiser produced a substantial improvement in

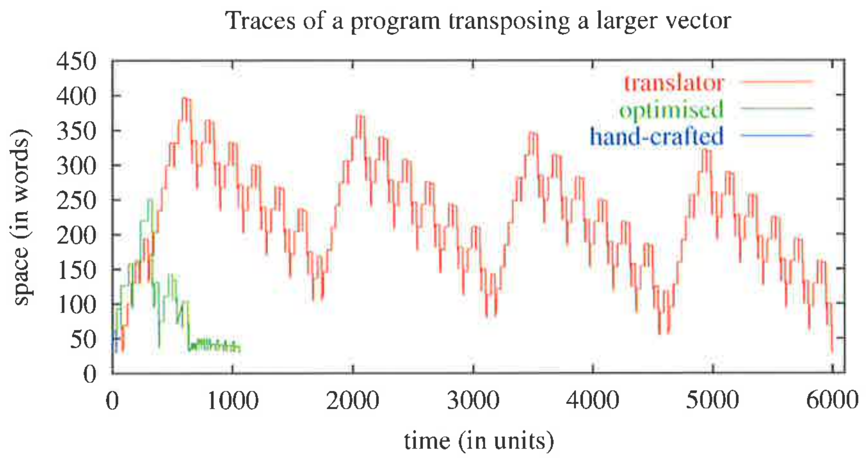


Figure 106. The relative performance of translator-code, hand-crafted code and optimised BMF code for `transpose.Adl` when applied to a larger (4×6) nested input vector.

performance. The precise extent of that improvement was dependent, for the most part, on the amount and efficiency of any redundant code, if any, left behind by the optimisation process. On balance, the optimisation process, defined so far, is very effective in many cases but there is scope for further enhancements to eliminate some of the redundant code remaining in some examples.

5.6.1.2 Other aspects of performance

Reliability Reliability is paramount in language implementation and even prototypes should exhibit some measure of reliability. Our implementation does, in the set of small examples we have tested it with, produce the same input to output mapping over a range of test data and, by close visual inspection, the code appears equivalent[37].

However, the optimiser stage occasionally loops or abnormally terminates on some more complex examples. As a general observation, loops can be remedied by making rules more mutually exclusive and/or capturing extra cases or by employing normalisation to reduce the number of cases. Abnormal termination is usually caused by the absence of an appropriate catch-all case or a simple error in a rule. Where

[37]The optimisation process is, in theory, capable of producing an audit trail for verification of its correctness. However, in our experience from tracing the proof process, such an audit trail is far too long to inspect manually. There is probably scope to build a tool to filter out the steps, deemed by the user over time, to be obviously correct to make inspection of such proofs tractable.

looping or abnormal termination have been encountered, we have, thus far, been able to remedy them.

Robustness Robustness is a measure of an implementations sensitivity to changes in its environment. The vector optimisation process and the tuple-optimisation process have different levels of robustness. The rules of the vector optimiser are geared precisely to the output produced by the translator. Any substantial changes to the code produced by the translator will adversely affect vector optimisation. Tuple optimisation is much more robust. Tuple optimisation makes few assumptions about code. The tuple optimiser is even able to process its own output[38]. In subsequent implementations of the optimiser we intend to employ assertions to codify the input requirements of the various rule-sets. Carefully constructed assertions will provide valuable guidance in the design of output from normalising rule-sets (e.g. *Remove_ids* and in designing rules catering for input to the transformative rule-sets (e.g *Opt*).

Time taken to compile and optimise programs The optimisation phase is easily the most computationally intensive part of the compilation process. Our prototype can run very slowly (in the order of minutes) when presented with moderately large programs.

There are two primary reasons for this slowness. The first is the amount of backtracking generated by the underlying Prolog implementation when the rule-sets are executed. This can be and has been, partially, remedied by re-designing rule-sets so that large proofs of premises are not undertaken when there is a good chance of failure in the normal course of compilation. Unfortunately such redesign appears to detract somewhat from the elegance and declarative style of natural semantic definitions. In any case future implementations are likely to be built in languages without backtracking so any performance penalty caused by backtracking will disappear.

The second and, ultimately, more important contributor to the slowness of the optimiser is the frequent use of normalisation. Casual observation of the optimiser in the Centaur debugging environment indicates that the optimiser spends a large part, if not the majority of its time performing normalisations such as associating

[38]Moreover, it appears, from the few observations we have made so far, that Tuple optimisation is an idempotent operation on its own output, converging to a fixed point after one iteration.

code to the left and removing identity functions. While there may be scope to apply normalisation more selectively, in some cases, it is unlikely that its role with diminish significantly in future implementations.

It remains to be seen whether slow compilation and long times spent performing normalisation is characteristic of implementations that use the transformational approach supported by BMF. A priori, BMF offers excellent scope for exploiting computationally intensive transformational strategies at compile time as part of a trade-off for producing faster target code. Whether such a trade-off is inevitable is an open question.

5.6.1.3 Elegance of target code

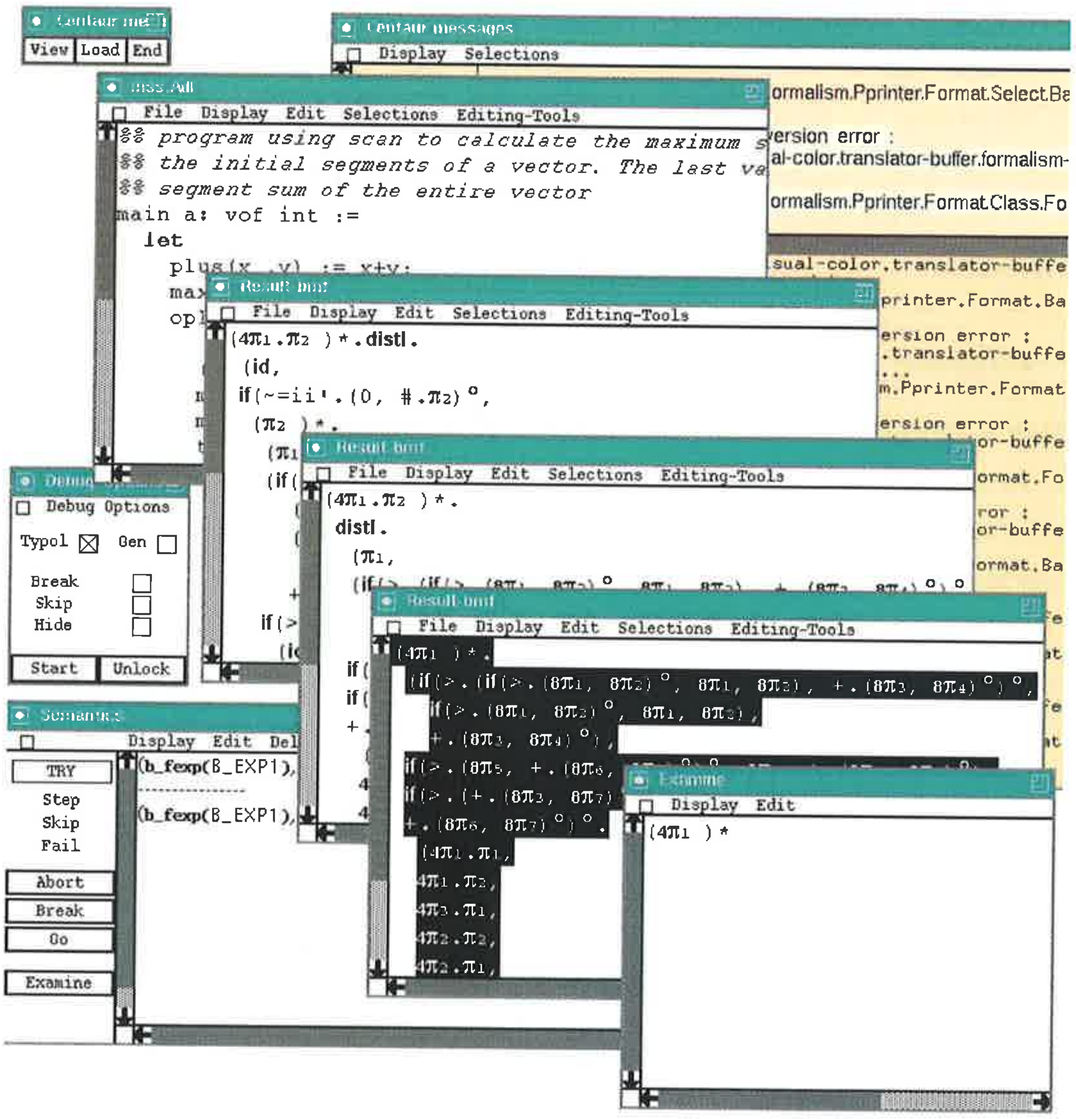
One of BMF's oft touted strengths is its elegance. It is possible to write quite powerful programs in a small amount of space using BMF notation[39]. Though it isn't of paramount importance when BMF is used as an intermediate form, elegance in the target code would seem, a-priori, to be a positive sign of convergence of form towards efficient handwritten code.

In our experiments to date our optimiser has, on occasion, produced code comparable in elegance and efficiency to hand-written code. However, the optimiser has also produced programs that, while reasonably efficient, have not been particularly elegant. Elegance, and efficiency, at least in our current implementation seem to be largely unrelated.

5.6.2 Findings related to building the optimiser definition

Centaur and Natural Semantics are a good medium for prototyping Using the Centaur system, we were able to create a prototype of the the optimiser at the same time as we defined its formal semantics. Definitions expressed in Natural Semantics have a clarity and elegance not easily emulated in most programming languages. Transcription of Natural Semantics definitions into Typol, the semantic notation underpinning Centaur definitions, was straightforward. The debugging, editing and display environment provided by Centaur (and shown in figure 107) was extremely useful for locating errors in our definitions.

[39]Though, we have tended to find that some of this elegance disappears once programs become detailed and we take away the option of reverting to other notations when convenient.



```
ormalism.Pprinter.Format.Select.Ba
ersion error :
al-color.translator-buffer.formalism-
ormalism.Pprinter.Format.Class.Fo
sual-color.translator-buffe
printer.Format.Ba
ersion error :
.translator-buffe
...
m.Pprinter.Format
ersion error :
or-buffe
ormat.Fo
ror :
or-buffe
ormat.Ba
```

```
%% program using scan to calculate the maximum s
%% the initial segments of a vector. The last va
%% segment sum of the entire vector
main a: vof int :=
  let
    plus(x ,y) := x+y;
    ma
    op
```

```
(4π1 . π2 ) * . distl .
 (id,
 if(~=ii' . (0, # . π2) °,
  (π2 ) * .
   (π1
   (if(
```

```
(4π1 . π2 ) * .
distl .
 (π1,
 (if(> . (if(> . (8π1, 8π2) °, 8π1, 8π2), + . (8π3, 8π4) °) °
```

```
(4π1 ) * .
 (if(> . (if(> . (8π1, 8π2) °, 8π1, 8π2), + . (8π3, 8π4) °) °,
   if(> . (8π1, 8π2) °, 8π1, 8π2),
   + . (8π3, 8π4) °),
 if(> . (8π5, + . (8π6,
 if(> . (+ . (8π3, 8π7)
 + . (8π6, 8π7) °) ° .
  (4π1 . π1,
  4π1 . π2,
  4π3 . π1,
  4π2 . π2,
  4π2 . π1,
```

```
(b_fexp(B_EXP1),
---------------
(b_fexp(B_EXP1),
```

```
(4π1 ) *
```

Figure 107. A screen dump of the Centaur system during translation and optimisation. Code at each stage of the implementation is given a separate window. Other windows are provided for debugging and viewing code bound to currently executed rules. Several pretty printers can be defined for source and target languages for various purposes.

Our initial prototype ran very slowly on larger programs. This slow performance was primarily due to a rule-structure oriented toward readability rather than optimal performance. We refined our definition to be more syntax-directed which greatly enhanced its performance. The refinement of a definition from one oriented towards a human reader to one oriented toward execution seems to be quite a natural development path. Peyton-Jones' and Meijer's article describing Henk[110] and Tullsen's article defining the Zip-calculus[136] both refine semantic definitions to be more oriented to an executable process.

Even after refinement, our optimiser rules, at least at the top level, are easily read. Unfortunately, this readability does not extend to all of the lower-level rules. Low level rules reflect the compromises taken to make an open-ended process such as program optimisation tractable. There is, perhaps, a mismatch between the practical concerns of optimisation and the customary purity of formal semantic definitions. However, on balance, the discipline, elegance, and clarity provided by a formal definition was to our advantage.

Incremental definition of optimiser rules is possible BMF is quite a forgiving medium for optimisation. Most BMF transformations are self-contained and localised. There are no large "all-or-nothing" transformations. This means that a partial implementation of an optimiser can selectively ignore parts of program code and still produce a correct, if not optimal, program.

Gaps in rules are costly Although it is advantageous to be able to work with partial definitions during construction of the optimiser, the gaps left by partial definitions can be costly in terms of performance of target code.

One cost is that any unprocessed section of code will have to be catered for by other parts of the optimiser, whose complexity may have to increase as a result. Another, more insidious, hazard of not catering for all cases occurs in rule-sets devoted to the normalisation of code. Missing cases in normalisation rules-sets mean that problematic code:

1. persists into the next transformative stage which, almost inevitably, will not have specific rules to handle it and

2. will probably persist, in perhaps a different guise, through future applications of the normalising rule-set.

The consequence is often that chunks of unoptimised code persist through to the end of the optimisation process. In the worst cases, these chunks accrete more code as optimisation progresses. Not surprisingly, this unoptimised code can severely impact the cost of the code ultimately produced by the optimiser.

Ultimately, temporary gaps are inevitable during any reasonable process of incremental development. It is important to be aware, during development, that their consequences are sometimes disproportionate to their size and, as a corollary, if at all possible, gaps need to be eliminated from the final implementation. To the best of our knowledge, all of the gaps that can be easily eliminated[40] have been eliminated from the normalisation rules of our implementation.

Care is needed to avoid infinite loops in the optimisation process Infinite loops are a potential hazard for transformative systems. Simple loops where an equality is applied backwards then forwards ad-infinitum, can be avoided by the simple expedient of applying rules only in one direction[47]. This heuristic can be thwarted by the creation of cycles between rules. To illustrate, the following two rules:

$$\frac{(f_1 \cdot f_2) \cdot f_3 \Rightarrow g}{f_1 \cdot (f_2 \cdot f_3) \Rightarrow g}$$

$$\frac{f_1 \cdot (f_2 \cdot f_3) \Rightarrow g}{(f_1 \cdot f_2) \cdot f_3 \Rightarrow g}$$

are mutually recursive and will cause the infinite loop:

$$\begin{array}{c} f_1 \cdot (f_2 \cdot f_3) \Rightarrow \\ (f_1 \cdot f_2) \cdot f_3 \Rightarrow \\ f_1 \cdot (f_2 \cdot f_3) \Rightarrow \\ (f_1 \cdot f_2) \cdot f_3 \Rightarrow \\ \cdots \end{array}$$

[40]Completely ridding all normalisation rule-sets of all gaps is more difficult than it might first seem. For example, the *Remove_ids* rule set has lacks a rule to detect that $\mathsf{select} \cdot (\mathsf{id}, \mathsf{iota} \cdot \#)^\circ$ is equivalent to id and can be eliminated where redundant. This absence is a gap but it is one that is difficult to eliminate completely, given the infinite number of variations on the theme of this code.

Ultimately, it is incumbent upon the author of the rules to avoid such cycles[41]. Some loops can be avoided by careful ordering of rules though the presence of backtracking can defeat this in some systems. One safe route is to normalise code so that a rule that otherwise might form part of a cycle is not needed. As a concrete example, the Adl optimisation process associates compositions to the left so that subsequent rules could avoid a cycle by having rules only for left-associated compositions.

Sundry observations on using point-free form The use of point-free BMF as a medium for automated program development is still an area of research. Observations relating to what works and, just as importantly, what doesn't work, are pertinent. Some observations follow.

Normalisation works In spite of the fact that it makes no direct contribution to efficiency, normalisation is pivotal to the Adl implementation. Normalisation is needed to lay the ground so that optimising rule sets can be effective without the need for complex analysis or clever use of cases. Normalisation creates a simple environment so that the rules that follow it can also be simple. Simpler rules are harder to get wrong. Note that this observation is simply an extension of what is already known about normalisation in other fields. Normalisation is, for example, used heavily in preparing statements in the propositional and predicate calculus for proof by resolution[33]. Of course, normalisation, often manually applied, is also heavily employed in relational databases to make it easier to maintain them in a consistent state[49].

Action-at-a-distance is difficult One drawback of point-free form is that it is very difficult to enact simultaneous changes to two spatially remote but related functions. This is not a trivial issue because optimisations such as common-sub-expression elimination can require such action-at-a-distance. This difficulty can be overcome in Natural Semantics by unifying one of the function with a logic variable than can then be acted upon in conjunction with the other function to be changed[42]. A-priori a more robust approach is to compact code in order to bring the parts to be

[41]Though one might envisage an automated system that could detect *some* such cycles.

[42]This technique is used by the vector optimiser during post-processing of map functions to re-generate addressing functions after tuple optimisation. These rules are some of many omitted from this chapter.

changed close enough that a simple re-write rule can be used. This approach, which we have not yet directly employed, is one that we will explore in future implementations.

Comparing addressing functions is difficult This issue is strongly related to the last issue. In point-free BMF, two functions addressing the same value can look different. This problem, where the name of a value is context-sensitive, is due to the fact that the addressing environment is new for each function. Again, a potential solution is to compact code to bring addressing functions into the same context if possible.

Pervasive transformation is easier To optimise any single part of a program the optimiser must traverse the code from the start or finish of the program to that point. Optimisation strategies that are able to perform useful work along the way are at an advantage. Both vector and tuple optimisation incrementally optimise code *as* they traverse it.

A better zip is needed Restricting target code to use ordinary binary zip introduces nested tuples where a flat tuple of larger arity would suffice. The clarity of code is compromised by such usage. The use of a form such as *transpose-tuple* from FP[145] or the generic zip used by Roorda in[121] is a likely refinement.

Binary composition is useful The associativity of function composition makes it feasible to define it as an operator on lists of functions rather than a simple binary operator. Though a list-version of the composition operator would make it easier to read BMF programs it is unlikely to be adopted to future Adl implementations. This is due to the fact that binary composition has proved to be such a useful tool in guiding the optimisation process. Binary composition allows the implementor to change the way that rule's match without materially affecting the semantics of the code.

This concludes this chapter's findings. A summary of work related to the Adl optimisers follows.

5.7 Related work

This section is divided into two parts. The first part describes work that achieves similar ends to the vector and tuple optimisers but in different domains. The second part puts the transformational method used by the optimiser into some context.

5.7.1 Optimisations related by goal

Research into optimisation of programs dates back to the first compilers [124] and several optimisation techniques were well established by the early 1970's[9]. Enumerating the optimisations that currently exist would take too long but Mahlke's Masters thesis[97] contains a good survey of classic optimisations.

Data-flow analysis The primary optimisation technique employed by the Adl optimiser is data-flow analysis[93, 32]. In a broad range of implementations, data-flow analysis is used to survey data dependencies throughout a program for purposes including: elimination of redundant code, removing invariant code from loops, detecting opportunities for parallel execution and even optimising data distribution. Data flow analysis is usually considered to be a form of global analysis because variable dependencies can span large parts of a program.

The data-flow analysis performed by the Adl optimisers differs from most because the code expresses data flows quite directly[43], and thus results of local analysis are, in most cases, integrated immediately into the code. This fine-grained alternation between analysis and transformation appears to be a natural way to use point-free BMF. The locality of transformations also gives a spatial dimension to the optimisation process with moving fronts of optimisation appearing in code and sweeping through the code. Lastly, in contrast with many implementations, the Adl optimisers have very small requirements for auxiliary data structures[44], changes are recorded directly in the code rather than in auxiliary structures.

Data movement optimisation In terms of their aims, analogues to data movement optimisation are likely to be found in domains where there is a significant, avoidable, *cost* attached to:

[43]There are no variables in point-free BMF to provide an implicit link between producers and consumers of data. Instead producers and consumers are linked directly through composition.

[44]Some of the structures that we do use may be eliminated in future versions.

1. Creating large, at least partially redundant, data structures.
2. Keeping such data accessible to later stages of a computational process.

The first point is a significant cost in many domains. The second is only of significance where data has to be copied or carried around in order to be accessible. Examples of optimisations tackling the first point include:

- query optimisation in databases by moving selects before joins to avert the creation of larger-than-necessary intermediate data structures[49](section 10.7).
- definition-propagation to prevent the formation of temporary values.
- Deforestation/Fusion[143, 108] to avoid the creation of large temporary data-structures.
- The judicious use of scalars in loops to avoid the creation of large temporary arrays in imperative programs.

There is a broad resemblance between vector optimisation and query optimisation. In vector optimisation the selection operation is brought into contact with the distribution operation to reduce the total amount of data created. The code-compaction carried out during tuple-optimisation is related to definition propagation insofar as they both reduce the number of intermediate values generated between a value's definition and its ultimate use. Loop fusion (a narrow specialisation of deforestation/fusion) is also a side-effect of code-compaction[45]. There is currently no Adl optimisation analogous to using temporary variables in loops as a substitute for storing whole arrays.

Examples of optimisations addressing point number two, the cost of keeping data accessible, are mainly to be found in parallel implementations[46]. In parallel systems there can be a high communications cost attached to keeping data accessible, and indirectly, to the creation of data structures with large amounts of redundancy. It follows that this field provides a source of optimisations to reduce such costs.

[45]Thus far work has made no direct attempt to employ other forms of fusion/deforestation to optimise programs further. Integrating existing techniques into this implementation is future work.

[46]The idealized cost of keeping data accessible throughout a program on a von-Neumann machine is zero. Of course, there is an actual cost in terms of making efficient use of the memory hierarchy but this is usually handled by a combination of the low-level parts of the compiler, and, of course, the hardware.

Related optimisations in parallel computing The need to explicitly capture the global environment for referencing is recognised, in a parallel context, by the designers of Algorithmic Skeletons[42]. In particular, the FARM skeleton can contain an explicit parameter to pass the referencing environment. This is very similar to the wrapping up of the environment with the input parameters to our map function in Adl translator code. Similarly, work in optimising IF1 data flow graphs by Feo and Cann[55] recognises the need to pass an environment to loop instances. Wu and Kelly stated the need to filter this global environment in their work on M-Trees [113].

Walinsky and Banerjee, in their implementation of FP*[145], recognise the need to reduce the data movement and copying costs. Their approach is to translate FP* code, a point-free form, to an imperative intermediate form, resembling CM-Fortran. They use *definition propagation* to track changes to data-structures wrought by *routing* functions[47]. These functions are subsequently eliminated and references in downstream code are adjusted to compensate. Our approach differs in two major aspects.

- We aim to retain explicit BMF routing functions at the distributed level to allow for consolidated communication of data required by nodes ahead-of-time rather than on-demand. Our strategy introduces the need to anticipate data needs of downstream code and optimise accordingly. This need is much less strong in the FP* implementation where data needs are satisfied on-demand[48].
- FP* restricts the placement of iota functions and avoids the use of functions as general as select in order to enhance the effectiveness of definition propagation[49]. Our implementation imposes no restrictions of this kind.

There are several other common optimisations used to reduce total communications costs in a parallel context, many of them work with distributed Fortran programs. Some of these are[68]:

[47]A routing function is a function that re-arranges, copies, or projects from, aggregate data without changing its constituent values.

[48]In the FP* implementation, the CM-Fortran compiler is responsible for partitioning and generating communications implicit in random access to implicitly distributed vectors.

[49]These restrictions might be lifted if some runtime infrastructure for mapping references to their data, in the fashion of the other stream of the Adl project[53], were provided by the target machine.

Message-Vectorization/Strip-mining: dragging independent statements, that trigger remote access to same node, outside of their enclosing loop and consolidating them into a single request for a vector of data.

Rationalising remote access in conditionals: by transmitting data that is used by both branches of a conditional only once and sharing it.

Moving Communications earlier: to subsume previous communication in order to consolidate/hide[50] latency costs.

The benefits of the message aggregation that message vectorization provided by the optimisations in the first point strongly motivate this work [51]. Under the parallel model described in the next chapter message aggregation occurs quite naturally, whether vector optimisation has taken place or not, however, without vector optimisation the size of these messages is typically much larger. The action of vector optimisation in pushing indexing functions outside of map functions to form a select is also strongly related to pushing individual indexing functions outside of a loop to form a vectorized communication command.

On the second point, there is no action in the vector optimiser corresponding to rationalising remote access in conditionals. However, the tuple optimiser does merge accesses to tuple elements in this way.

On the third point, there is no attempt to merge/subsume communications in the vector optimiser. However, the tuple optimiser trivially subsumes redundant access to elements of tuples in this way. There are no current plans to exploit latency-hiding in this thread of the Adl project, though there are no fundamental impediments to doing so if the need arises.

The vector optimiser's use of a select to consolidate statically unknowable patterns of communication is mirrored in von Hanxleden, Kennedy, et al.[141] and, in later work by on array remapping operations by Dietz, et al.[48]. Hanxleden, Kennedy, et al.'s paper focused on hoisting piecemeal array access to one-dimensional arrays out of loops in Fortran-D into equivalent *gather*[52] and *scatter* operations. This

[50]Latency hiding is implemented by issuing a non-blocking receive before the data is actually needed. The corresponding send has time to occur and then, when the data is actually needed, it may already have arrived, if it hasn't the process can wait until it does.

[51]These benefits are widely known[14] and are well exploited by models such as BSP[76]

[52]Gather has almost identical semantics to the parallelised code for select, which is described on page 209 in chapter 6.

process is complicated by the imperative semantics of Fortran and the approach is, necessarily, conservative. The paper by Dietz, et al. focuses on optimisation of remapping constructs that specify *gather* and *scatter* operations over regular distributed multi-dimensional arrays in the language ZPL[107]. Their work exploits shape and distribution information found in *region* specifiers associated with values, which helps support aggressive optimisation of the implementations scatter and gather operations. Note that, in ZPL, the programmer is responsible for expressing consolidated communications operations through remapping constructs, whereas in Adl it is the job of the optimiser to do this.

Related optimisations for re-oriented vector access The problems caused by a mismatch in the order of loop index variables relative to the indexing functions embedded in these loops are well understood[45]. Adl vector optimisation solves this problem using transpose to re-orient the nested vector to better match the indices generated for the surrounding map functions, thus making it possible, in most cases, to move indexing functions outward and consolidate them into selects. This contrasts with established loop-transformation techniques which leave the vector as-is and swap the nesting of loops to achieve the same (or similar) ends. The use of transpose rather than loop-swapping, by the vector optimiser, is partly motivated by desire to keep transformations local. However, with the appropriate normalisation, loop-swapping transformations might be made local, and these could be an attractive alternative to using transpose.

Finally, it is important to mention that some other implementations reduce communications costs by accessing distributed data on demand. Several implementations including Id[115], SISAL[55], Lucid[11], and the multi-threading, thread of Adl project[54] use demand-driven data-access, among other techniques, to minimise the volume of communication. The trade-off is increased overhead from many small messages[53].

This concludes the summary of optimisations related by goal. Next, the transformative techniques used by both Adl optimisers are put into some context.

[53]Note that it is still possible to aggregate some requests for data in demand-driven system as long as the mechanism for aggregation precludes deadlock. The potential advantages of such a system warrant further exploration in the context of the Adl project.

5.7.2 Optimisations related by technique

There is a large body of work on optimisation of functional programs via program transformation. Early work includes the `fold-unfold` transformations developed by Burstall and Darlington (citations from this part can be found in [123]). Other types of mechanisable or partially mechanisable transformation include[108, 78]:

- *tupling* to consolidate multiple recursive calls into one.
- *diffusion* turning recursive definitions into non-recursive BMF.
- *fusion/deforestation* merging composed functions and, more generally, eliminating intermediate data structures[143].
- *flattening* avoiding possible load imbalance due to nesting of higher-order BMF functions.
- *partial-evaluation* to specialise calls to some functions with respect to static knowledge of their input values[104].

Our current implementation does not exploit any of the transformations above though there are no impediments to exploitation of these in future Adl implementations[54]

A complicating factor in any process of automatic transformation is the presence of recursive definitions which can render the composition of otherwise correct transformation steps unsound[123][55]. Bird[18] circumvents these problems by using a calculus (BMF) of composed functions, similar in style to Backus' FP[13] but with a strongly typed foundation based in category theory[131], as a medium for transformation. In BMF, recursive definitions are typically not exposed at the level at which transformation takes place so the hazards of composing rewrites do not occur. Later work in this field (see [15, 63, 38] for a sample) has shown that such an approach can indeed lead to efficient programs and, perhaps more importantly, to valuable insights into some problems.

There are a broad spectrum of possibilities for harnessing programming transformation techniques as a compilation tool. Table 2 summarises the possibilities. Each row of the table represents a type of optimisation. The meanings of the columns

[54]A lifting of the current embargo against recursive definitions in Adl would open the way for full exploitation of existing diffusion techniques.

[55]This article provides an excellent characterisation of these problems as well as techniques for circumventing them.

Optimisation	Prob domain	Purpose	Extent	Rule-choice	Rule-set	Granularity
Ideal	arbitrary	general	global	automatic	self-extensible	fine
User-assisted	arbitrary	general	global	assisted	extensible	fine
Annotated	arbitrary	general	local function	limited auto-matic	user extends	fine
Vector/Tuple optimisation	non-recursive	specific	global	static	static	fine
Local opts	varied	specific	local	static/ arbitrary	static	fine
Global opts	varied	specific	broad	static	static	typically coarse

Table 2. A characterisation of different types of optimisation according to a range of criteria.

are:

Prob domain The nature of the programs this type of optimisation can be applied to. For the purposes of exposition, we assume that all optimisations are restricted to functional programs[56]. Ranges down from "arbitrary" where any functional definition can be optimised to various kinds of restrictions.

Purpose The purpose of the optimisation. All other things being equal, general purpose optimisation is harder to do than specific-purpose optimisation.

Extent The spatial extent of the optimisation. Ranges down from "global" to "local".

Rule-choice How the rules of the optimiser are applied. Ranges down from "automatic", where the optimiser decides which rules to apply dynamically, to "static" where the author of the optimisation decides which rules to apply.

Rule-set How the rule-sets that govern the optimisation process are composed. Ranges from "self-extensible" where the optimiser extends its own rule-set to "static" where the author of the optimisation determines the rule-set.

Granularity The granularity of transformation. Ranges from "fine" to "coarse". Fine-grained changes are incremental. Coarse-grained changes are large, often

[56]Some optimisation techniques are also applicable to imperative programs.

opaque, all-or-nothing transformations. All other things being equal, a fine granularity is preferred for its greater transparency.

The first three rows of the table contain general-purpose optimisations. The first row characterises an ideal optimisation, able to choose and generate rules needed to decrease execution costs for arbitrary programs and apply these rules in a fine-grained and transparent manner. Such systems do not yet exist. An ideal optimisation is unlikely to be effective without extra information as to how the program is going to be used[57]. The second row describes optimisations where the user-assists in the optimisation process. Martin and Nipkow[98] described this type of optimisation for BMF code by assisting a theorem prover. The third row describes optimisations that are guided by programmer annotations. The MAG system developed by de Moor and Sittamaparam[47] uses a simple heuristic to choose from rules, provided by the programmer, for each candidate function definition, to effect optimisation.

The last three rows describe optimisations directed to a specific purpose. The fourth row describes the vector and tuple optimisation phases of the Adl compiler. Both of these phases have a specific purpose and, indeed, they consists of even more specialised sub-phases. The impact of each optimisation is global and pervasive[58]. The choice of rule is statically determined by the implementors. The rule-set is also static, but perhaps more easily extensible than the rules defining more conventional approaches. The transformational granularity is fine[59]. Mottl's system for partial evaluation of definitions in a strict functional language might also be said to fall into this category. Rule choice is governed by the structure of code which is annotated so as to apply only to rules that will preserve this structure. Through these code-annotations, rules can be applied in an order which leads to termination of the

[57]Extra information like this is not to be underrated. Extra information about the shape of data can greatly contribute to efficiency [86]. Systematic observations of actual program performance have been successfully applied in incremental compilation[94]. Observations of actual program performance while it is running lead to very effective '*hotspot*' optimisations in Java Virtual-Machines[102]. Notably, the extra information takes away a lot of the analytical burden from the optimiser. Less information has to be deduced if you already have it to hand.

[58]As mentioned previously, the nature of point-free form seems to make it easier to define rules to perform pervasive transformations than to define rules that simultaneously act on spatially separate sections of code.

[59]One could argue, because the Adl optimisers compose transformations into groups, and because there are no choice-points based on cost functions, that the granularity is actually coarse. However, the transparency of the Adl optimisers, the lack of large steps, and the fact that they can be stopped at almost any point and the result is still a correct program, all work in favour of calling them fine-grained.

transformation process. The rules of the Adl optimisers use function composition in a similar way. Adl optimiser rules often change the associativity of composition in a monotonic way so that, at some point, only a "finishing rule" of a rule-set will apply which, effectively, returns the result of the transformation. Another optimisation with a similar flavour is Onoue et al's [108] system to automatically perform fusion in recursive functions in Gofer[88].

The last two rows capture most common forms of optimisation. Local optimisations are fine-grained transformations that effect improvement at a local level. Local optimisations are not assumed to be coordinated with each other and it is this that distinguishes them from the previous four rows[60]. An example of a local transformation might be the rule:

$$f * \cdot g* \Rightarrow (f.g)*$$

which fuses two maps. A large number of local transformation rules have been developed for BMF. Some transformations have been usefully applied in a distributed parallel context[62, 63, 79].

Global optimisations are those which require some sort of non-localised analysis such as data-flow-analysis. Global optimisations tend to be coarse grained and opaque. Often, they employ an abstraction outside of the language such as a data-flow graph or a polytope[45] and, if successful, they apply the optimisation in a wholesale manner. This contrasts strongly with the Adl optimisers which incrementally approach the point where cost-reducing transformations can be applied[61]. Compilers often employ a battery of global optimisations to cover different aspects of performance.

Fine-grained vs Coarse-grained As a final note, it is worth briefly comparing fine-grained and coarse-grained approaches to optimisation. As mentioned previously, all other things being equal, fine-grained optimisation is to be preferred due to its

[60]The distinction can be very fine. Philip Wadler's set of rules for concatenate elimination[142] can, individually, be viewed as local transformations but together they form a system where the rules are applied to a program, until they can be applied no more. There is nothing in Wadler's rules that dictates an order of application, the system is robust enough to cope with an arbitrary order of application (it is confluent). So we have a system but it is very loosely coordinated.

[61]Note that the use of localised rewrite rules does not prevent effective global optimisations from being achieved. Visser's excellent survey of applications of rewriting strategies[140] aptly illustrates this point.

transparency and potential flexibility. However, not all things are equal. For example, if fine-grained transformations are applied in a way that monotonically reduces cost at each step, a local minimum can be reached. Coarse-grained transformations avoid this problem by leaping past local minima in a single step. To be effective, fine-grained transformation systems must sometimes make programs worse before making them better[62]. The statically defined rules of the Adl optimisers achieve this by normalising code, without immediate regard to efficiency, to a point where significant cost-saving transformations *can* be applied[63]. Non-static systems must employ some other technique that take account of global costs in order to avoid local minima. Skillicorn et. al[129] describes a technique that utilises global costings to effectively optimise BSP style programs.

This concludes our summary of related work. A brief summary of the chapter and proposals for future work are given next.

5.8 Summary and Future Work

This chapter described the optimisation process applied to the translator code of the previous chapter. The process consists of two phases: vector optimisation, to reduce the transport of individual elements contained in vectors, and tuple optimisation to reduce the transport of individual elements contained in tuples. Both phases of optimisation work by a process of incremental transformation that maintains the correctness of the program during optimisation. The optimisation process takes place with very limited resort to any auxiliary data structures.

Both phases of optimisation place a heavy reliance on normalisation to reduce the number of cases that subsequent rule-sets need to cater for. The most challenging part of the optimisation process is catering for arbitrary combinations of vector references inside of map functions. Optimisation rule sets are amenable to incremental construction though it was observed that the omission of specific rules to handle each function in normalising rule-sets were costly in terms of performance of target code.

[62]Elliot, Finne and de Moor[52] found that the freedom to make things worse before making them better paved the way for some very useful optimisations in the implementation of their domain-specific language PAN.

[63]In an intuitive sense this can be likened to climbing up to, and going over, the edge of a very large cliff in the cost-space.

Finally, it was shown that the optimisation process is highly effective when applied to a range of programs, in some cases, producing code of comparable performance to hand-written BMF.

5.8.1 Future work

There are a number of directions for future work on the optimiser definition. A good first step would be to rewrite the optimiser in a way that avoids the overheads of backtracking. It is likely that, as part of this process the optimiser definition will be ported to a new platform[64]. In addition, a new version of the optimiser could be instrumented to produce a trace of a proof as it applies its rules. A cost model could be used to annotate the intermediate code produced by the process.

There is scope for more use of normalisation and more concrete definitions of the input interfaces for non-normalising rule-sets and output interfaces for normalising rule-sets. Interfaces of this kind have been usefully employed in rewriting-based implementations in Stratego[46, 138].

Though vector and tuple optimisation warrant separate consideration, the idea of interleaving these phases within a single pass to produce cleaner wish-lists bears investigation.

There is some scope to apply further transformations to increase the efficiency of code based on techniques related to deforestation[143] (to fuse composed operations) and tupling[80] (to avoid multiple traversals of a data structure).

Currently, we limit the, non-normalising transformations used, to those that invariably increase the efficiency of code. A cost model could be used to guide the application of transformations whose benefit is dependent on certain conditions[65]. The eventual goal is to incrementally improve the optimisation process with the addition of passes to implement various optimisation techniques as they are discovered.

The optimiser has succeeded in increasing the efficiency of BMF programs produced by the translator. The next chapter presents a process by which these programs can be efficiently mapped to distributed-memory architectures.

[64]Most likely to be Haskell[81] or Stratego[139].

[65]Some characterisation of the input data will be required to decide whether or not to apply some such transformations.

Chapter 6

Parallelisation and Targetting

To this point, this report has shown how a simple functional language can be converted to point-free BMF code [1] and then shown how the efficiency of that code can be dramatically improved through the automated application of a series of normalisation and optimisation steps. This chapter describes a process for parallelising sequential BMF, describes how this parallel BMF might then be mapped to a distributed computer and then explores performance issues that can arise.

The work we describe here corresponds to the last two stages of the implementation shown in figure 108. Prototypes for these stages are in the early stages of development[146, 111]. The techniques described here form the basis of these prototypes.

Structure of this chapter A brief preview of our approach to parallelisation and code generation is presented next. After this, we present the parallelisation process applied to a few important constructs followed by an overview of the code-generation process and the issues that it raises. At the end of this chapter we present performance results for a series of simple applications run under various conditions on a simulator and, finally, related literature is reviewed.

6.1 Preview

Parallelisation and code generation present major design choices.

[1]As with other chapters we will henceforth, in this chapter use the term BMF to mean *point-free BMF code.*

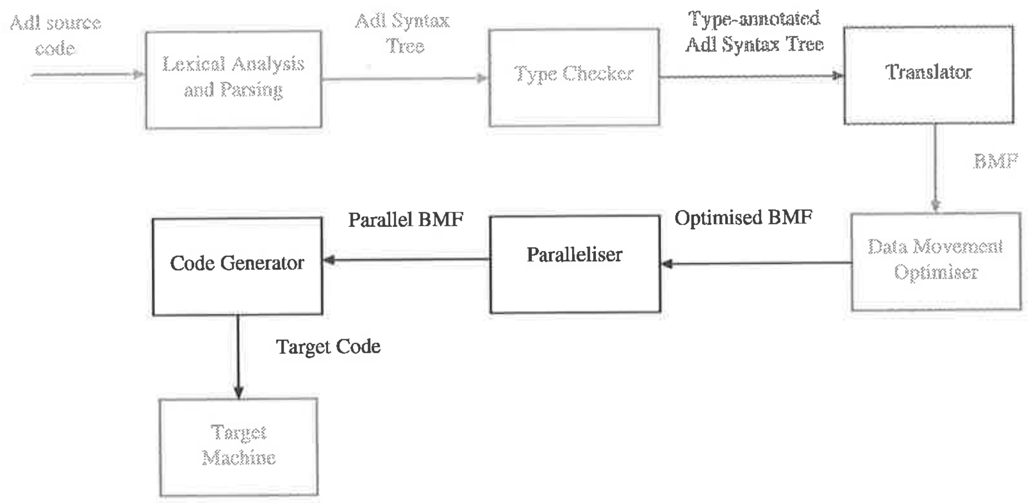


Figure 108. The Adl project with the parallelisation and code-generation marked in black.

- For parallelisation, these choices revolve around how explicit to make the parallelism in the code.
- For code generation, we must choose the extent to which we rely on a run-time system to manage the efficient execution of applications.

We now briefly discuss the choices we have made for these two stages.

6.1.1 Design choices for parallelisation

There is a spectrum of design choices for the parallelisation process. The two extremes of this spectrum are:

Fully implicit: Use a sophisticated run-time system to exploit the parallelism implicit in unmodified BMF code.

Fully explicit: Define a stage of compilation to embed most aspects of parallelism in target code.

We briefly discuss these extremes and our design choice next.

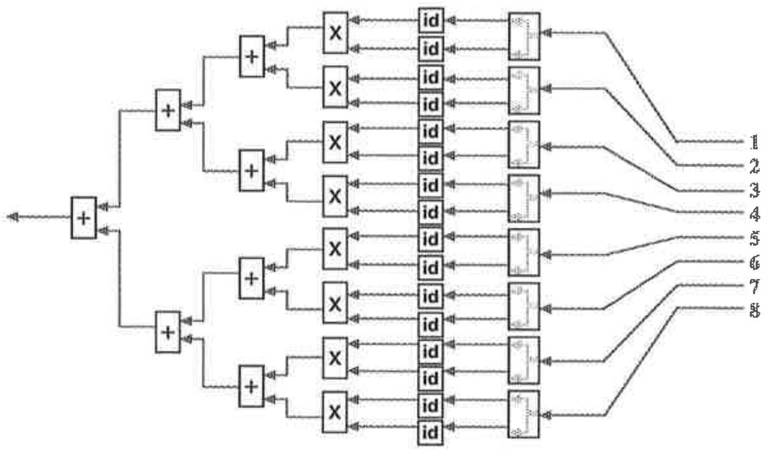


Figure 109. Graph of data paths for: $+/_0 \cdot (\times \cdot (\mathsf{id}, \mathsf{id})^\circ\ [1,2,3,4,5,6,7,8]$

Implicit parallelism Once we know the dimensions of its input data a BMF program is easily unrolled into a data-flow graph. For example, the graph for the expression:

$$+/_0 \cdot (\times \cdot (\mathsf{id}, \mathsf{id})^\circ) * \ [1,2,3,4,5,6,7,8]$$

is represented in figure 109. With a run-time system able to dynamically allocate virtual-processors there is much parallelism that can be exploited both at the level of vector functions such as map and within alltup functions. There a two main drawbacks to such a system:

1. The overheads of the run-time system including the maintenance of the mapping between virtual processors and real processors.
2. Performance penalties from carte-blanche application of parallel algorithms on virtual processors.

The second point is very significant. For some operations such as scan the optimal parallel implementation requires the use of an efficient sequential algorithm on each node followed by the application of a different parallel algorithm between nodes. The use of virtual processors *hides* the boundaries between real nodes which prevents the use of such hybrid algorithms. In summary optimal performance requires that asymptotically optimal algorithms can be expressed in the target code and it is a fact that some of these optimal algorithms have distinct parallel and sequential parts.

Explicit parallelism Explicitly defined parallelism avoids many of the run-time overheads of implicit parallelism and provides scope to define algorithms optimised for the chosen partitioning scheme. The costs of explicit parallelism are:

1. The need for an additional stage in the compiler to inject parallel constructs into optimiser code.

2. A potential lack of responsiveness to varied input and/or varied runtime conditions.

The first cost is just the trade-off between increased compilation time and better runtime efficiency of applications. The second cost arises because, with explicit parallelism, at least some decisions affecting performance are made at compile-time. Such decisions are, necessarily, based on incomplete information which can lead to sub-optimal performance if run-time conditions are not those expected by the writer of the compiler.

Our design choice for parallelisation In our implementation, we have chosen to use an automated process to embed explicitly parallel constructs in optimiser code. The parallelisation phase of the compiler is given the optimised BMF program and an integer denoting the number of processing nodes on which to map the program. The paralleliser then statically embeds the decomposition of data, mapping of computations and data to processing nodes and most aspects of communication in the target program it generates. The routing of messages generated by the parallelised select function is handled at run-time.

Next, we briefly examine possible design choices for code generation.

6.1.2 Design choices for code generation

Design choices for code generation are determined by issues such as:

1. The amount of information available at compile time.

2. The amount of work we expend writing the code-generation module.

Addressing the first point, our implementation assumes that code emanating from the parallelisation process provides enough information to produce code that requires minimal interpretation at runtime. In particular, we assume that little or no type information has to be carried around at runtime. Using static type information the Adl implementation monomorphically instantiates calls to polymorphic functions (such as map[2]) for the sake of efficiency.

Addressing the second point, the code generation phase, could be very simple or quite complex. Unsurprisingly, as a first implementation, we use a simple approach to code generation. There is scope to improve efficiency using further analysis[3]. We revisit the topic of code-generation in more detail in section 6.2.

6.1.3 Summary of our design choices

In both parallelisation and code generation we have opted to make as many decisions at compile-time as possible. Parallelisation embeds nearly all aspects of parallelism either implicitly or explicitly in the code. Code generation embeds this information directly in the target form. Now, we examine these two processes in turn - starting with parallelisation.

6.2 Parallelisation

Parallelisation is the process of converting a sequential BMF program into a semantically equivalent parallel BMF program. Details of how this is done for each construct are given later, but first we provide an overview by means of an example.

6.2.1 Overview of the parallelisation process

A short example Given the the program to calculate the sum of squares in figure 110, the paralleliser produces the code in figure 111. The functions with the superscript: $||$ have the same meaning as the ordinary function with the same name but with parallel semantics. For example, $f*^{||}$ is an application of the map function

[2]Kuchen [95] took the approach of defining polymorphic library functions for such constructs. This allows applications to exploit improvements in these constructs' implementation without recompilation.

[3]The cost of memory management, in particular, can be addressed by deeper analysis at this point.

$$+/ \cdot (\times \cdot (\mathsf{id}, \mathsf{id})^\circ)*$$

Figure 110. Simple BMF program to calculate the sum of squares of an input vector.

$$+/^{||} \cdot (+/) *^{||} \cdot ((\times \cdot (\mathsf{id}, \mathsf{id})^\circ)*) *^{||} \cdot \mathsf{split}_p$$

Figure 111. Parallelised sum of squares program.

with simultaneous invocations of the function f on distinct processing nodes. The current implementation parallelises vector operations. The split_p function has a special role. Its purpose is to convert a vector into a distributed vector of vectors[4]. The number of sub-vectors is determined by the subscript p which, at this stage, is assumed to be a parameter supplied by the user of the compiler. split_p also attaches a closure for a *partitioning function*[5] to the descriptor of its output array. The partitioning function accepts a single integer, representing an index to a value into the original input vector of split, and returns a pair of integers which represent the indices of the corresponding value in the nested output vector of split[6]. The partitioning function is attached to the type of the distributed vector and resides on the *parent-node.* The parent-node is the processing node from which the split function was invoked. The means by which a partitioning function can be extracted and applied are described later in this chapter.

Now the components of the example program have been introduced. Figure 112

[4]The *unfold* function, described by Gibbons[60] has split as a special case.

[5]This function fulfils a similar role to the partitioning function used in other aspects of the Adl project[54]. SCL (Structured Coordination Language) [43] also uses a partitioning function but this is explicitly provided by the user. Currently Adl avoids such an explicit function because it provides scope for non-contiguous partitionings that can compromise the use of associative but non-commutative functions in reduce and scan.

[6]For example, if:

$$\mathsf{split}_3\,[1, 2, 3, 4, 5, 6, 7]$$

produces the nested vector:

$$[[1, 2, 3], [4, 5, 6], [7]]$$

The partitioning function pf for this vector could be represented as:

$$pf\ x = (\lfloor x/3 \rfloor, x\,\mathsf{mod} 3)$$

$$\begin{array}{ll}
+/^{\|} \cdot (+/) *^{\|} \cdot ((\times \cdot (\mathsf{id}, \mathsf{id})^{\circ})*) *^{\|} \cdot \mathsf{split}_3\, [1,2,3,4,5,6,7] & \Rightarrow \\
+/^{\|} \cdot (+/) *^{\|} \cdot ((\times \cdot (\mathsf{id}, \mathsf{id})^{\circ})*) *^{\|}\, [[1,2,3],[4,5,6],[7]] & \Rightarrow \\
+/^{\|} \cdot (+/) *^{\|}\, [[1,4,9],[16,25,36],[49]] & \Rightarrow \\
+/^{\|}\, [14,77,49] & \Rightarrow \\
140 &
\end{array}$$

Figure 112. Evaluation of parallel sums of squares applied to $[1, 2, 3, 4, 5, 6, 7]$

shows steps in the parallel evaluation of the program in figure 111 with $p = 3$ and with input $[1, 2, 3, 4, 5, 6, 7]$. The result of this evaluation is exactly the same as if it had been carried out sequentially but computations at the outer level of the nested vector are carried out in parallel.

The new distributed semantics Thus far, we have shown how the input-output mapping of a program can be preserved during parallelisation. Now we examine how the execution of the parallelised program has changed. The original sequential program operates within the confines of a single processing node. In contrast, each parallelised function has a distributed semantics.

Figure 113 shows how the distributed computation evolves during the execution of the parallel program from figure 111. Step 0 shows the program, data and machine prior to execution. The box on the right represents a single node (node 0) with the data on it. Step 1 shows the state of the program, its data and the machine after the execution of split_3. The sub-vectors reside on the child-nodes 0.0, 0.1, and 0.2. Step 2 shows the state after the parallel application of the inner map. Step 3 shows the state after the parallel application of the inner reduce. Step 4 shows the finishing state after the application of the parallel reduce.

Note that the split and parallel reduce functions trigger communication. The split is a special case of a *scatter* where contiguous blocks of a vector are distributed across a set of processing nodes. The reduce is responsible for gathering the results back on to a single node, aggregating intermediate results using the $(+)$ function. The intervening map operations cause no communication.

Nodes are numbered in a hierarchical manner. Child node numbers are suffixed with their parent node number. The hierarchy can be extended to describe a nested distribution. Note that nested parallelism is not the prime focus of this work but it is

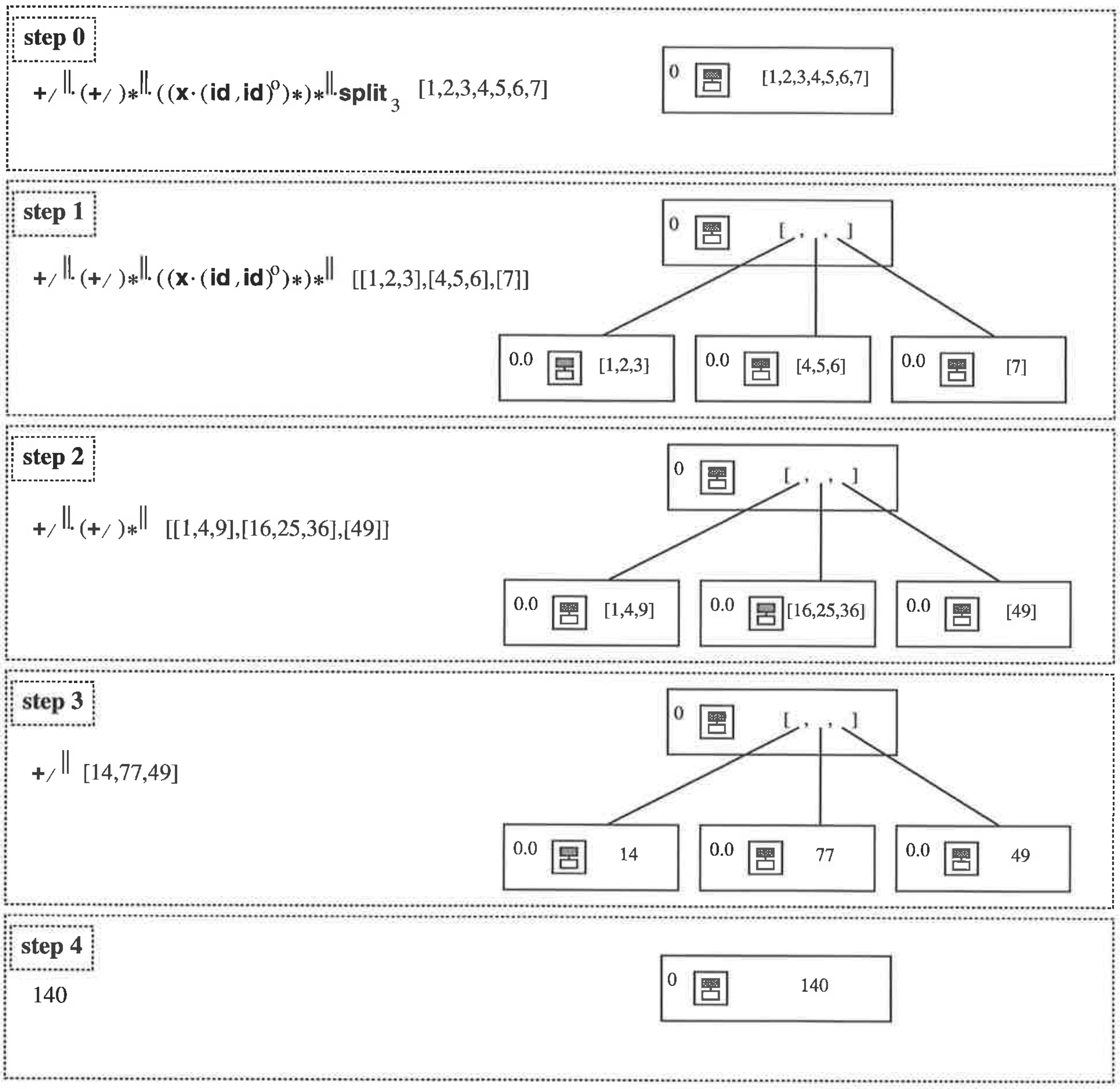


Figure 113. Sequence of steps in evaluating parallel sum-of-squares applied to the vector $[1, 2, 3, 4, 5, 6, 7]$

beneficial to allow scope for the expression of regular or irregular nested parallelism in future. A preliminary discussion of nested parallelism is in appendix E.

Now we have seen a specific example of the parallelisation process and its execution model. A general overview of the parallelisation process follows.

General overview of the parallelisation process In keeping with the approach to compilation we have described thus far, parallelisation occurs by applying a series of incremental, localised, semantics-preserving, transformations. Essentially, given a block of sequential BMF code consisting of a sequence of composed functions with each producing a vector:

$$h \cdot g \cdot f \cdot e$$

the parallelisation process inserts, *downstream* of the code, a special identity function on vectors. The function is defined as $+\!\!+ /^{||} \cdot \mathsf{split}_p$. The $+\!\!+ /$ in this identity function is a parallel reduce with concatenate. This function flattens a distributed list back onto the parent node. The code with the special identity function in place is now:

$$+\!\!+ /^{||} \cdot \mathsf{split}_p \cdot h \cdot g \cdot f \cdot e$$

The split_p function is then pushed up upstream by a process of transformation:

$$\begin{array}{ll}
+\!\!+ /^{||} \cdot \mathsf{split}_p \cdot h \cdot g \cdot f \cdot e & = \\
+\!\!+ /^{||} \cdot h' \cdot \mathsf{split}_p \cdot g \cdot f \cdot e & = \\
+\!\!+ /^{||} \cdot h' \cdot g' \cdot \mathsf{split}_p \cdot f \cdot e & = \\
+\!\!+ /^{||} \cdot h' \cdot g' \cdot f' \cdot \mathsf{split}_p \cdot e & = \\
+\!\!+ /^{||} \cdot h' \cdot g' \cdot f' \cdot e' \cdot \mathsf{split}_p &
\end{array}$$

where, depending on their structure, h', g', f', and e' may be modified at deeper levels. An intuitive reading of the process is that, because split_p distributes a vector p ways and $+\!\!+ /^{||}$(or any other instance of parallel reduce) gathers it, any operation sandwiched between the two is distributed and, of course, distributed operations may execute in parallel. The parallelisation process proceeds from the downstream rather than the upstream side to make it easier to ensure matching distributions for vectors with a common destiny. Figures 114, and 115 help illustrate this point. Figure 114 abstractly illustrates the problem caused by parallelising from the wrong, upstream, side of the program. The two converging lines represent two vectors that are destined to be zipped together at some later point in the program. If parallelisation proceeds

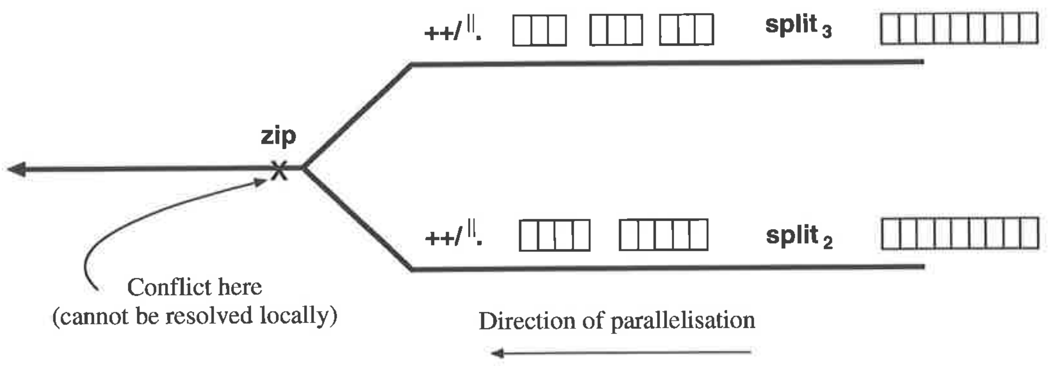


Figure 114. Two vectors, distributed in different ways at an earlier point of the parallelisation process can come into conflict when they are joined by a function such as zip. Resolving the conflict will require at least one of the instances of split to be changed.

from the upstream side then it is possible to distribute the two vectors differently. Later on in the parallelisation process the zip, which requires matching distributions on its inputs, is encountered. It is awkward at this stage to go back and change one of the split functions so that it matches the other.

Figure 115 shows how parallelising from the downstream side avoids this problem. Part (a) shows the state of the program just prior to the parallelisation of zip. At this point there is only one vector so there can only be one distribution. In part (b) the zip function has been parallelised and each of its two input vectors have been given the same distribution. Here, there is no conflict to resolve.

Parallelising from the downstream side avoids conflict where two vectors share the same destiny but what about the converse case where two vectors share the same *origin*? Figure 116 shows how parallelisation from the downstream direction copes with this case. Part (a) shows how two vectors, initially given different distributions, are found to have the same origin. Part (b) shows how this conflict is resolved by choosing one of the two distributions. Note that there is no requirement to change already-parallelised code as there was in figure 114[7].

For the few examples shown to date, all functions in the program have vector inputs and results. What happens if functions have non-vector input or output types? These cases must be handled with various rules that allow parallelisation to permeate

[7]This asymmetry of requirements for changing stems from an asymmetry between split_p and $+\!\!+/^{\|}$. A split_p must be changed if the required distribution of its *output* changes. In contrast, there is no need to change $+\!\!+/^{\|}$ if its *input* distribution changes.

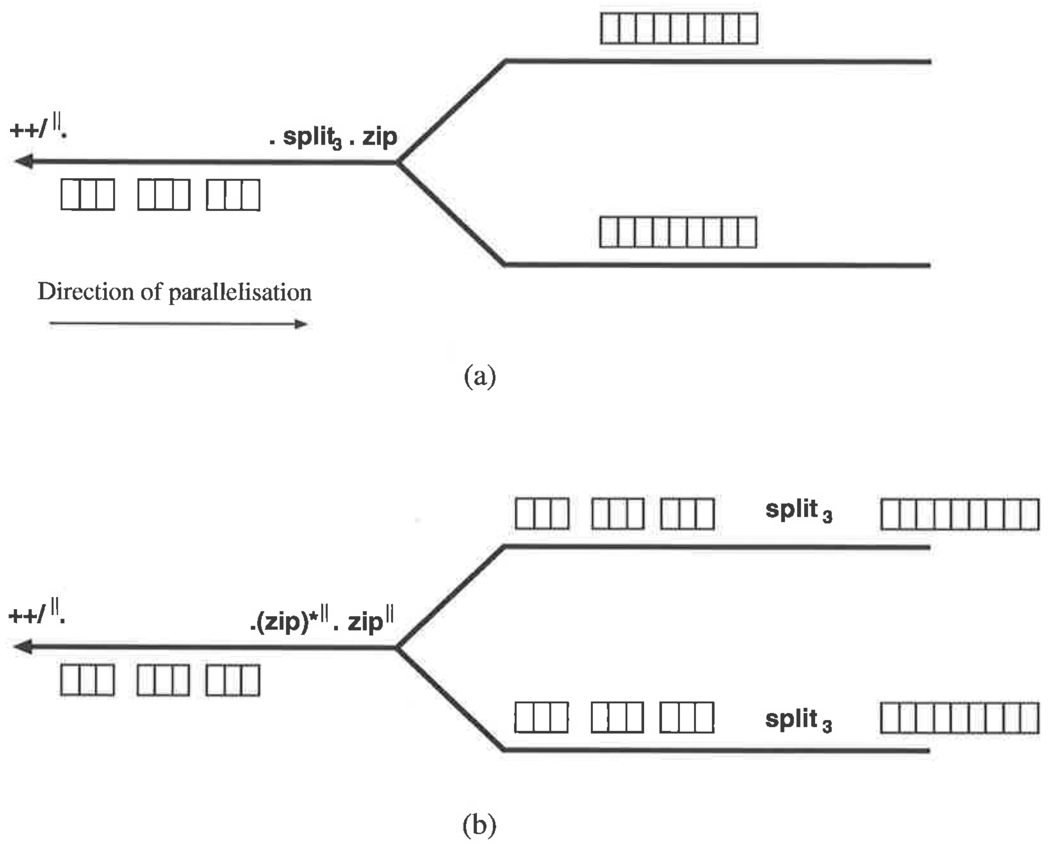
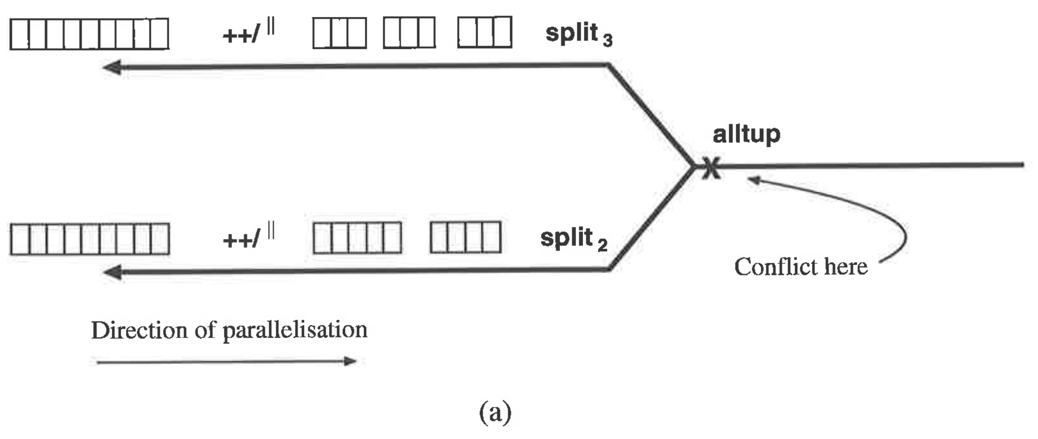


Figure 115. Conflicting distributions caused by common destination can be avoided by parallelising from the downstream side. Two vectors that are destined to meet (part (a)) will share a single distribution (part (b)).

down to any vector components in these data types. It must be noted that not every function of every program is amenable to parallelisation. Where a segment of code cannot be parallelised it is left unaltered and the parallelisation process recommences immediately upstream. It should also be noted that not every amenable function is worth parallelising under all circumstances. The current process is quite eager and will parallelise even when the pay-offs are small or non-existent. Future versions could avoid this by using a cost model such as that described by Skillicorn and Cai[128] to evaluate the benefits of each step of parallelisation.



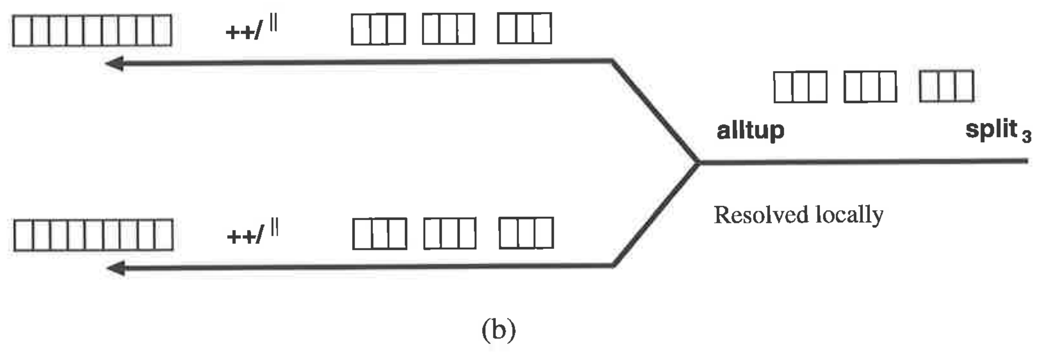


Figure 116. Parallelisation from the downstream side can result in conflicting distributions when two vectors share a common origin (part (a)). The conflict is easily resolved by a local decision at the common point of origin (part (b)).

6.2.2 The rules of parallelisation

This section lists parallelisation rules for most[8] of the BMF syntax produced by the optimiser. In all cases, rules are expressed in terms of moving split_p through the function from the downstream side. Where it might be helpful, the meaning of a function is briefly recapitulated just prior to the rule. New parallel functions are briefly described as they are introduced. The discussion surrounding each is kept as informal as possible.

A note on types As with previous stages of compilation, rules must preserve and maintain type information. It is assumed that every function is annotated with a

[8]If a function is not explicitly handled a default rule applies. This rule, given on page 215, skips the next function and resumes parallelisation upstream of that function. This helps keep the process correct and robust.

$$\frac{\begin{array}{c} \mathit{Associate\_Right}(\mathit{prog} \cdot \mathsf{sentinel} \Rightarrow f_{\alpha \to \beta} \cdot \mathit{prog}') \\ \mathit{Gen\_Suffix}(\beta \Rightarrow s_1 \cdot s_2) \\ \mathit{Parallelise}(s_2 \cdot f \cdot \mathit{prog}' \Rightarrow \mathit{prog}'' \cdot \mathsf{sentinel}) \end{array}}{\mathit{prog} \to s_1 \cdot \mathit{prog}''}$$

Figure 117. The top level rule for the parallelisation process

monomorphic description of its type. As with data-movement-optimisation, many parallelisation rules are not dependent on explicit type information. To keep the discussions brief, types are only shown when the operation of parallelisation rule depends on them.

Now to the rules. There is one rule defining the top level of the paralleliser and this is described first. The rules for the primary rule-sets invoked by this rule follow.

6.2.2.1 Top rule

The top level rule for parallelisation is shown in figure 117: The first premise composes a sentinel function to the front of the program and then associates all occurrences of function composition in the resulting program to the right[9]. The function f is the most downstream function in the right-associated program. The sentinel function exists to ensure that there is at least one function from which to create *prog'*. The sentinel will never appear in the target program so its semantics are unimportant. However its type can affect the second premise of this rule so, for consistency, we set the output type of the sentinel function to be the same as the input type of the program.

The call to *Gen_Suffix* in the second premise generates a *suffix* function for the program based on the output type of f. For example, if the output type of the

[9]The use of *Associate_Right* is yet another application of normalisation. Again the the immediate aim is not to achieve efficiency but predictability for the sake of subsequent transformations. *Associate_Right* converts an expression with arbitrary associativity of function compositions such as:

$$(f_1 \cdot (f_2 \cdot f_3)) \cdot (f_4 \cdot f_5)$$

to an equivalent expression with composition operators uniformly associated to the right:

$$f_1 \cdot (f_2 \cdot (f_3 \cdot (f_4 \cdot f_5)))$$

As with *Associate_Left* the transformation applies recursively to all sub-expressions containing composed functions.

function f is a vector then *Gen_Suffix* will generate the function $+\!\!+/ \cdot \mathsf{split}_p$. The suffix function always consists of two components:

1. the upstream function, s_2, which uses split_p functions to distribute any vector data in the output of f. In the explanations to follow call s_2 the *distributor*.

2. the downstream function, s_1, which uses $+\!\!+/$ functions to re-coalesce any distributed vector data it receives as input. In the explanations to follow we call s_1 the *gatherer*.

Composed together, the two components of the suffix functions form an identity function on whatever input type they are given. The role of the suffix function is to serve as a launch-pad for the parallelisation process invoked in the third premise. This parallelisation process takes the distributor, s_2, and pushes it as far upstream through $f \cdot prog'$ as possible, leaving distributed code in its wake.

The conclusion of the rule composes $prog'$, the code produced by the parallelisation process, with the gatherer function s_1 to form the final result of the program.

The *Gen_Suffix* and, of course, the *Parallelise* rule-sets warrant further discussion and we introduce these in turn.

6.2.2.2 Generating the suffix

Figure 118 shows the rules for *Gen_Suffix*. The rules all take a type and generate a suffix function from which to start the parallelisation process. The syntax for types was given on page 50, in chapter 4. We show the more concrete version, used in figure 118, in figure 119[10].

Each of these rules generates an appropriate suffix function for a given type. Rules 31, 32, and 33 handle functions of scalar type which present no immediate opportunities for parallelisation. The suffix produced by these rules consists of paired identity functions: $\mathsf{id} \cdot \mathsf{id}$.

Rule 34 is key, it creates the suffix $+\!\!+/ \cdot \mathsf{split}_p$ to serve as a base for parallelisation of vector functions upstream.

[10]Note that the phyla *B_TYPE* contains no functional type. This is because, in the programs produced by our implementations, functions cannot be treated as values. This restriction is relaxed very slightly with the introduction of partitioning functions on page 203 but not in a way that affects the workings of the paralleliser.

Set *Gen_Suffix* **is** $\boxed{B\_TYPE \Rightarrow B\_EXP}$

Integers

$$\texttt{integer} \Rightarrow \mathsf{id}_{\texttt{integer}\to\texttt{integer}} \cdot \mathsf{id}_{\texttt{integer}\to\texttt{integer}} \quad (31)$$

Reals

$$\texttt{real} \Rightarrow \mathsf{id}_{\texttt{real}\to\texttt{real}} \cdot \mathsf{id}_{\texttt{real}\to\texttt{real}} \quad (32)$$

Booleans

$$\texttt{boolean} \Rightarrow \mathsf{id}_{\texttt{boolean}\to\texttt{boolean}} \cdot \mathsf{id}_{\texttt{boolean}\to\texttt{boolean}} \quad (33)$$

Vectors

$$[\beta] \Rightarrow +\!\!+ /^{||}_{[[\beta]]\to[\beta]} \cdot \mathsf{split}_{p\ ([\beta]\to[[\beta]])} \quad (34)$$

Tuples

$$\frac{\begin{array}{c} \beta_1 \Rightarrow (f_{11})_{\gamma_1\to\beta_1} \cdot (f_{12})_{\beta_1\to\gamma_1} \\ \mathit{Prefix\_Addr}(f_{11}, {}^n\pi_1 \Rightarrow f'_{11}) \\ \mathit{Prefix\_Addr}(f_{12}, {}^n\pi_1 \Rightarrow f'_{12}) \\ \cdots \\ \beta_n \Rightarrow (f_{n1})_{\gamma_n\to\beta_n} \cdot (f_{n2})_{\beta_n\to\gamma_n} \\ \mathit{Distribute\_Addr}(f_{n1}, {}^n\pi_n \Rightarrow f'_{n1}) \\ \mathit{Distribute\_Addr}(f_{n2}, {}^n\pi_n \Rightarrow f'_{n2}) \end{array}}{(\beta_1, \ldots \beta_n) \Rightarrow (f'_{11}, \ldots, f'_{n1})^\circ \cdot (f'_{12}, \ldots, f'_{n2})} \quad (35)$$

End *Gen_Suffix*

Figure 118. Rules for generating a suffix function.

$$\begin{array}{lll} B\_FTYPE & ::= & B\_TYPE \to B\_TYPE \\ B\_TYPE & ::= & \texttt{integer} \mid \texttt{real} \mid \texttt{boolean} \mid \\ & & B\_TVAR \mid [\ B\_TYPE\ ] \mid (B\_TYPE, \ldots, B\_TYPE) \\ B\_TVAR & ::= & \alpha \mid \beta \mid \gamma \mid \ldots \end{array}$$

Figure 119. Syntax of BMF types

Rule 35 handles tuple types. It is the most complex rule in *Gen_Prefix* and warrants a deeper description. Tuples may contain a mixture of scalar, vector and tuple types and the suffix function produced by this rule must accommodate all of these. As an example of how this rule works, given the type:

$$(\texttt{integer}, [\texttt{integer}])$$

it generates the suffix function:

$$(\mathsf{id} \cdot \pi_1, +\!\!+ / \cdot \pi_2)^\circ \cdot (\mathsf{id} \cdot \pi_1, \mathsf{split}_p \cdot \pi_2)^\circ$$

The two id functions apply to the `integer` value in the first element of the input tuple. The $+\!\!+ /$ and split_p function access the vector forming the second element of the input value. In the example above, the π_1 functions ensure that the scalar data is routed through the id functions and the π_2 functions ensure that vector data is routed through the split_p and $+\!\!+ /$ functions.

As a second example, given the type:

$$((\texttt{integer}, [\texttt{integer}]), \texttt{real})$$

the tuple rule generates the suffix function:

$$((\mathsf{id} \cdot \pi_1 \cdot \pi_1, +\!\!+ / \cdot \pi_2 \cdot \pi_1)^\circ, \mathsf{id} \cdot \pi_2)^\circ \cdot ((\mathsf{id} \cdot \pi_1 \cdot \pi_1, \mathsf{split}_p \cdot \pi_2 \cdot \pi_1)^\circ, \mathsf{id} \cdot \pi_2)^\circ.$$

Again, the π_1 and π_2 functions are used to route values from the input through the appropriate parts of the distributor and gatherer functions. Note that where the generated alltup functions are nested more than one addressing function is required in the prefix.

Looking at the tuple rule in detail, we see that the premises recursively invoke *Gen_Suffix* on each element i of the tuple type to produce a suffix function of the form:

$$f_{i1} \cdot f_{i2}$$

In the conclusion of the rule each of these functions must be embedded in separate alltup functions and data must be routed appropriately from f_{i2}, the distributor function, to f_{i1}, the gatherer function, and also from code upstream to f_{i2}. This routing is achieved by prefixing appropriate address functions to both of these functions. This prefixing is carried out by the *Prefix_Addr* rule-set, defined in

Set *Prefix_Addr* **is** $\boxed{B\_EXP, B\_EXP \Rightarrow B\_EXP}$

Alltup functions

$$\frac{\begin{array}{c} f_1 ,^m \pi_i \Rightarrow f'_1 \\ \ldots \\ f_n ,^m \pi_i \Rightarrow f'_n \end{array}}{(f_1,\ldots,f_n)^\circ ,^m \pi_i \Rightarrow (f'_1,\ldots,f'_n)^\circ} \qquad (36)$$

Other functions

$$f ,^m \pi_i \Rightarrow f \cdot^m \pi_i \qquad (f \neq (f_i,\ldots,f_n)^\circ) \qquad (37)$$

End *Prefix_Addr*

Figure 120. Rules for prefixing addressing functions components of a suffix function.

figure 120. The effect of *Prefix_Addr* is to propagate the address function into the most deeply nested parts of the alltup function downstream.

Rule 36 handles alltup functions by recursively calling *Prefix_Addr* on each component function of the alltup. Rule 37 applies to all non-alltup functions. This rule simply composes the function with the required addressing function.

This completes our description of *Gen_Suffix* and *Prefix_Addr*. We now describe the *Parallelise* rule set, which defines the core of the parallelisation process.

6.2.2.3 The Parallelise rule-set

The *Parallelise* rule-set is responsible for replacing all sequential code acting on non-distributed vectors in its input program with parallel code acting on distributed vectors in its output program. As with the stages of optimisation seen in the previous chapter, this process is carried out by propagating the required transformations from the most downstream parts of the program to the most upstream parts. Descriptions of the rules for the *Parallelise* rule-set follow.

The sentinel function Figure 121 shows the rule that applies to code composed with a sentinel function. A sentinel function marks the most-upstream part of the program, which is the point at which the parallelisation process stops. The rule returns its input values unchanged.

Set *Parallelise* **is** $\boxed{B\_EXP \Rightarrow B\_EXP}$

The sentinel **function**

$$f \cdot \mathsf{sentinel} \Rightarrow f \cdot \mathsf{sentinel} \tag{38}$$

Figure 121. Parallelisation rule for handling the sentinel function.

Function Composition

$$\frac{\begin{array}{c} s \cdot f \Rightarrow f' \cdot s' \\ s' \cdot g \Rightarrow g' \cdot s'' \end{array}}{s \cdot (f \cdot g) \Rightarrow f' \cdot g' \cdot s''} \tag{39}$$

Figure 122. Paralleliser rule for handling function composition

Function Composition All sequences of composed functions have already been associated to the right by the first premise top level rule in figure 117. The pre-processing performed by this premise allows all sequences of composed functions to be processed by the single rule defined in figure 122. The first premise recursively applies the parallelisation process to the function f with respect to s, the distributor function. The result is the transformed code f' and the distributor function s'. The second premise applies the same process to the remainder of the composition sequence: g. The final version of the distributor function s'' is integrated into the composition sequence in the rule's conclusion[11].

Alltup alltup functions apply each of their constituent functions to the input value producing a tuple of corresponding result values. The paralleliser rules for alltup functions is given in figure 123. Rule 40 applies when the distributor function is an alltup function. The rule parallelises f with respect to each of the components of the distributor function producing a composed pair of alltup functions as a result. These result functions may contain common sub-expressions and it is future work to define rules to eliminate these.

Rule 41 applies when the function to be parallelised:

$$(f_1, \ldots, f_n)^\circ$$

[11]The one exception to this statement is if $f \cdot g$ is the outermost composition sequence in the program. In this case s'' will be a sentinel function rather a distributor function as a result of the application of rule 38.

Alltup distributor

$$\frac{\begin{array}{c} s_1 \cdot f \Rightarrow f'_1 \cdot s'_1 \\ \ldots \\ s_n \cdot f \Rightarrow f'_n \cdot s'_n \end{array}}{(s_1,\ldots,s_n)^\circ \cdot f \Rightarrow (f'_1 \cdot^n \pi_n, \ldots, f'_n \cdot^n \pi_n)^\circ \cdot (s'_1,\ldots,s'_n)^\circ} \qquad (40)$$

Alltup target

$$\frac{s \cdot f_i \Rightarrow f'_i \cdot s'}{s \cdot^n \pi_i \cdot (f_1,\ldots,f_n)^\circ \Rightarrow f'_i \cdot s'} \qquad (41)$$

Figure 123. Paralleliser rules for handling alltup functions

Addressing functions

$$s \cdot^n \pi_i \Rightarrow \mathsf{id} \cdot (s \cdot^n \pi_i) \qquad (42)$$

Figure 124. Paralleliser rule for handling addressing functions

is an alltup function. The rule's premise selects and transforms f_i, the addressed function, with respect to s, the downstream part of the distributor function. This rule assumes that the distributor function has some address function, ${}^n\pi_i$, as its most upstream function. The validity of this assumption is maintained, initially, by *Gen_Suffix* and, later, by other the rules of *Parallelise* including the previous rule (rule 40) and a rule for addressing functions, described next[12].

Addressing functions Figure 124 shows the rule for tuple-addressing functions. This rule simply sweeps the addressing function into the distributor function leaving an id function behind. Very often, the addressing function will be eliminated later by rule 41, the second rule for alltup functions[13].

Allvec functions An allvec function applies an input value to each of its constituent functions, producing, a vector of results. Figure 125 contains the rule for allvec functions. The current version of the rule does not allow the suffix function s to pass

[12]The previous rule maintains the validity of the assumption by stripping away any surrounding alltup function in the distributor to, eventually, reveal any leading address functions. The next rule allows distributor functions to assimilate address functions from upstream code

[13]The exceptions occur when the input type of the program is a tuple and the addressing function is accessing a part of that tuple. In these cases the elimination will not occur and the address

Allvec functions

$$\frac{\begin{array}{c} \mathit{Gen\_Suffix}(f_1 \Rightarrow s_{11} \cdot s_{12}) \\ \dots \\ \mathit{Gen\_Suffix}(f_n \Rightarrow s_{n1} \cdots_{n2}) \\ s_{12} \cdot f_1 \Rightarrow f'_1 \cdot s'_1 \\ \dots \\ s_{n2} \cdot f_n \Rightarrow f'_n \cdot s'_n \end{array}}{s \cdot [f_1, \dots, f_n]^\circ \Rightarrow (s \cdot [s_{11} \cdot f'_1 \cdot \pi_1, \dots, s_{1n} \cdot f'_n \cdot \pi_n]^\circ) \cdot (s'_1, \dots, s'_n)^\circ} \quad (43)$$

Figure 125. Paralleliser rule for handling allvec functions

Vector-indexing functions

$$\mathsf{id} \cdot ! \Rightarrow ! \cdot (!^{\Vert} \cdot (\pi_1, \pi_1 \cdot \pi_2)^\circ, \pi_2 \cdot \pi_2)^\circ \cdot (\pi_1, \mathsf{app} \cdot (\mathsf{pf} \cdot \pi_1, \pi_2)^\circ)^\circ \cdot (\mathsf{split}_p \cdot \pi_1, \pi_2)^\circ \quad (44)$$

Figure 126. Paralleliser rule for handling vector indexing functions

through the allvec function[14]. The first premises of the rule generate a new suffix function for every function in the allvec. In the second set of premises, the distributor component of these suffix functions are pushed through each of the component allvec functions to produce new transformed functions f'_i and new suffix functions s'_i. These two sets of components are combined in allvec and alltup functions respectively, in the rule's conclusion.

Vector indexing function The vector indexing function takes a pair, consisting of a vector and an integer, and produces the value from the vector found at the index represented by the integer. The rule for indexing functions is shown in figure 126. The code produced by this rule rule relies upon the application of a partitioning function to the original index value. The pf function takes a distributed vector as input and produces the closure of the partitioning function as output. The type of pf is:

$$[[\alpha]] \rightarrow (\mathtt{integer} \rightarrow (\mathtt{integer}, \mathtt{integer}))$$

functions will remain in the final code.

[14]This is an exception to the convention that all vector functions are parallelised. This exception is made because of the typically small vector size produced by a allvec function and the likelihood of a processing imbalance generated by the heterogeneous functions it contains.

Length functions

$$\mathsf{id}_{\texttt{integer}\rightarrow\texttt{integer}} \cdot \# \Rightarrow +/^{||} \cdot (\#) *^{||} \cdot \mathsf{split}_p \tag{45}$$

Figure 127. Paralleliser rule for handling length functions

Map functions

$$\mathsf{split}_p \cdot f* \Rightarrow f * *^{||} \cdot \mathsf{split}_p \tag{46}$$

Figure 128. Paralleliser rule for handling map functions

which makes it the only function in this implementation that returns a function as a result. The app function takes a partitioning function and an integer and applies the integer to the partitioning function to produce a pair of integers representing the two indices used to access the appropriate element of the distributed vector. Note the innermost index function on the right-hand side of the rule has a || superscript to indicate that it operates on distributed data.

Length The length function takes a vector and returns an integer representing the length of that vector. The rule for the length function is shown in figure 127. The code produced which is, effectively, a statement that the length of the original vector is the sum of the lengths of all of the sub-vectors of the distributed vector. Note that, by itself, length is rarely worth parallelising. However, often it does make sense to apply this rule if the parallelised length function is at the end of a long sequence of other parallel code[15].

Note that the rules for both index and length create a new distributor function: split_p.

Map The rule for map shown in figure 128. This rule adds a layer of nesting to the original map function to account for the distribution of its input vector. The outer map is parallel. This rule is a well-established identity in BMF[131].

Reduce The rule for reduce functions is shown in figure 129. This rule converts a

[15]In his report on the prototype implementation of the paralleliser, Windows' [146] noted that such an assessment is probably best made during an additional pass through the code.

Reduce functions

$$\mathsf{id}_{\alpha\to\alpha}\cdot\oplus/\Rightarrow\oplus/^{\|}\cdot(\oplus/)*^{\|}\cdot\mathsf{split}_p \tag{47}$$

Figure 129. Paralleliser rule for handling reduce functions

Scan functions

$$\mathsf{split}_p\cdot\oplus/\!/\Rightarrow\quad g\cdot(\oplus/\!/^{\|}\cdot\mathit{init}\cdot\mathit{last}*^{\|},\mathsf{id})^{\circ}\cdot(\oplus/\!/)*^{\|}\cdot\mathsf{split}_p \tag{48}$$

Figure 130. Paralleliser rule for handling scan functions

flat reduction, with a binary operator $\oplus$, over a flat vector into a nested reduction, with a binary operator $\oplus$, over a nested vector. This rule is another well-established BMF identity.

Scan There are several options for parallelising scan. The rule we use, producing an efficient parallel scan, is shown in figure 130. The parallel code on the right-hand-side of the rule was derived by Roe[120], for MIMD architectures. Where:

$$\begin{aligned}
&\mathit{last} = !\cdot(\mathsf{id},-\cdot(\#,1)^{\circ})^{\circ}\\
&\mathit{init}\ =\ \mathsf{select}^{\|}\cdot(\mathsf{id},\mathsf{iota}\cdot-\cdot(\#,1)^{\circ})^{\circ}\\
&g = +\!\!+^{\|}\cdot(\mathsf{split}_1\cdot!\cdot(\pi_2,0)^{\circ},(\odot)*^{\|}\cdot\mathsf{zip}^{\|}\cdot(\pi_1,\mathit{tail}\cdot\pi_2)^{\circ})^{\circ}\\
&\mathit{tail}\ =\ \mathsf{select}^{\|}\cdot(\mathsf{id},(+\cdot(1,\mathsf{id})^{\circ})*\cdot\mathsf{iota}\cdot-\cdot(\#\cdot\mathsf{id},1)^{\circ})^{\circ}\\
&\odot = (\oplus)*\cdot\mathsf{distl}\cdot(\pi_1,\pi_2)^{\circ}
\end{aligned}$$

The definitions of *init* and *last* are reasonably straightforward. The workings of the rest of the parallelised code can be illustrated by example. Given the input vector:

$$[1,2,3,4,5,6,7,8,9,10,11]$$

and $\oplus = +$ and $p = 4$ the parallelised scan code upstream of g produces:

$$([6,21,45],[[1,3,6],[4,9,15],[7,15,24],[10,21]])$$

where the second element of the tuple is a vector of scanned vectors. The first element $[6,21,45]$ is the result of applying parallel scan to the vector of initial last elements

$[6, 15, 24]$. The function g pairs the first vector with the tail of the second vector to produce:

$$[(6, [4, 9, 15]), (21, [7, 15, 24]), (45, [10, 21])]$$

the first elements of these pairs are distributed over the second elements:

$$[[(6, 4), (6, 9), (6, 15)], [(21, 7), (21, 15), (21, 24)], [(45, 10), (45, 21)]]$$

and added pairwise:

$$[[10, 15, 21], [28, 36, 45], [55, 66]]$$

by $(\odot)*^{\|}$. Finally, this result is concatenated with the initial sub-vector to produce:

$$[[1, 3, 6], [10, 15, 21], [28, 36, 45], [55, 66]]$$

note that the concatenate function $+\!\!+^{\|}$ is a distributed communication primitive to append one distributed vector to another[16]. Figure 131 gives a pictorial representation of the workings of the function g.

Distl distl takes a pair of the form $(v, [x_0, \ldots, x_{n-1}])$ and produces a vector $[(v, x_0), \ldots, (v, x_{n-1})]$. The rule for distl is shown in figure 132. The new parallel $\mathsf{distl}^{\|}$ function distributes a value from the parent processor over a distributed vector on the child processors.

Zip The rule for zip is shown in figure 133. The most upstream code, the new distributor function: $(\mathsf{split}_p \cdot \pi_1, \mathsf{split}_p \cdot \pi_2)^\circ$, produces a pair of distributed vectors. The pair exists at the level of the parent node but the sub-vectors exist in the child nodes. $\mathsf{zip}^{\|}$ pairs these sub-vectors at the level of the child nodes, as illustrated in figure 134. The most downstream function: $(\mathsf{zip})*^{\|}$ pairs the values within each of the sub-vectors.

Repeat Repeat takes a pair (v, n) and produces a vector of the form $[v, v, \ldots, v]$ of length n. The paralleliser rule for repeat is shown in figure 135. This rule creates a non-distributed vector of the required length using iota and distributes it, using split. The value to be repeated can then be distributed over this vector using the

[16]This function, like split_p has the ability to allocate more processing nodes. However, in this example, this ability is not required, the length of the result of $+\!\!+^{\|}$ is the same as the length of the distributed input vector for which nodes have already been allocated.

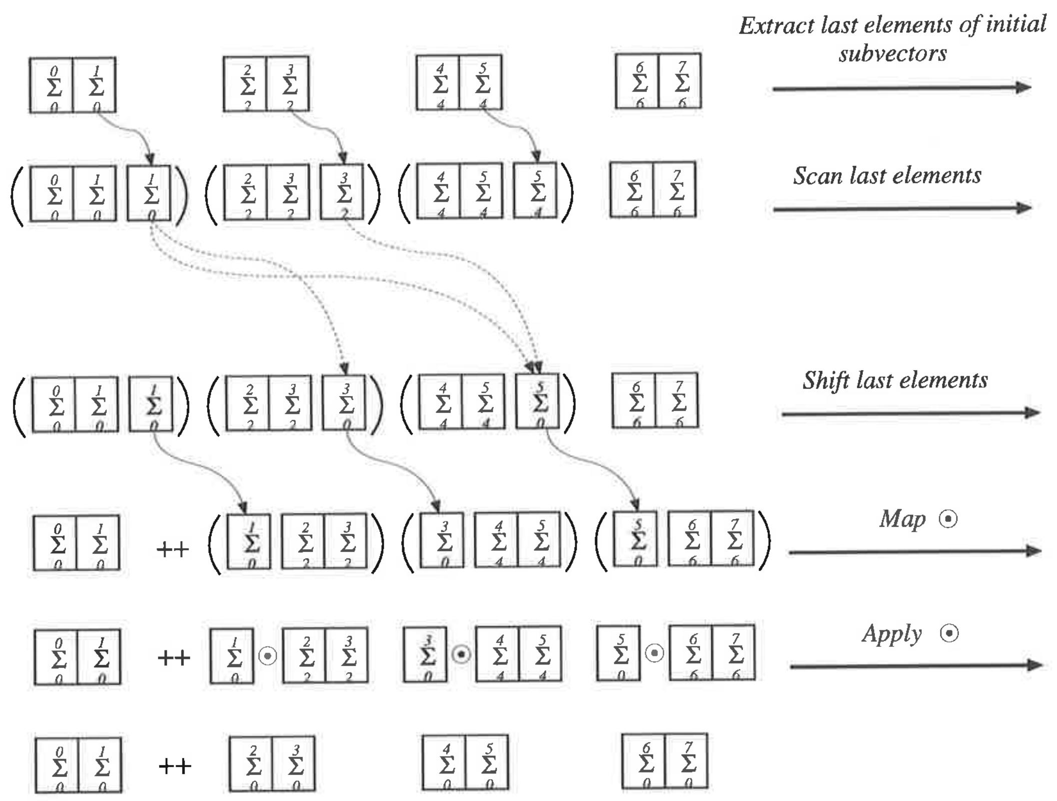


Figure 131. Using the scanned last values of sub-vectors (step 3) to propagate a summation of sub-vectors (step 4).

Distl functions

$$\mathsf{split}_p \cdot \mathsf{distl} \Rightarrow (\mathsf{distl}) *^{\|} \cdot \mathsf{distl}^{\|} \cdot (\pi_1, \mathsf{split}_p \cdot \pi_2)^\circ \tag{49}$$

Figure 132. Paralleliser rule for handling distl functions

Zip functions

$$\mathsf{split}_p \cdot \mathsf{zip} \Rightarrow (\mathsf{zip}) *^{\|} \cdot \mathsf{zip}^{\|} \cdot (\mathsf{split}_p \cdot \pi_1, \mathsf{split}_p \cdot \pi_2)^\circ \tag{50}$$

Figure 133. Paralleliser rule for handling zip functions

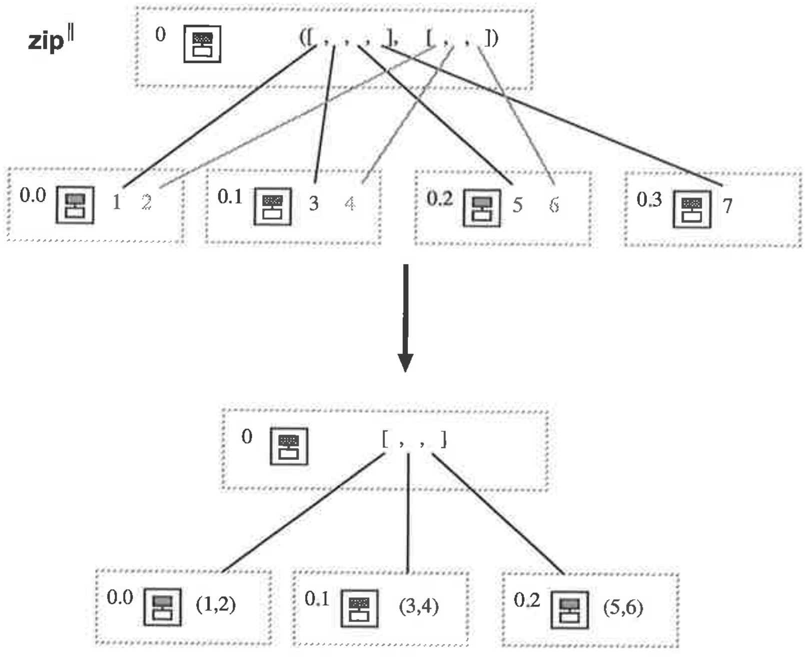


Figure 134. The effect of a zip operation over corresponding elements of distributed vectors.

Repeat functions

$$\mathsf{split}_p \cdot \mathsf{repeat} \Rightarrow (\pi_1) **^{\|} \cdot (\mathsf{distl}) *^{\|} \cdot \mathsf{distl} \cdot (\pi_1, \mathsf{split}_p \cdot \mathsf{iota} \cdot \pi_2)^\circ \qquad (51)$$

Figure 135. Paralleliser rule for handling repeat functions

parallelised code for distl. Finally, the $(\pi_1) **^{\|}$ function projects the required value out of the nested vector of pairs produced by the earlier code.

This code is not optimal, partly because of this final projection, but mostly because split needlessly consumes bandwidth by distributing an array, which is the length of the final result. If the final result is large compared to the number of processing nodes[17] a much less bandwidth-intensive solution is simply to distribute one copy of the value to each node and then to repeat the value within each node. The problem here is to achieve the desired partitioning. This can be done if the rule introduces a new primitive:

$$\mathsf{split}_p \cdot \mathsf{repeat} \Rightarrow \mathsf{distrepeat}_{(\alpha,\mathtt{integer})\to[[\alpha]]}$$

distrepeat that produces a nested vector along with its partitioning function. Our

[17]A very likely scenario.

Select functions

$$\begin{array}{l}\mathsf{split}_p \cdot \mathsf{select} \Rightarrow \\ \quad \mathsf{distselect} \cdot (\pi_1, (\mathsf{app}) * *^{\|} \cdot (\mathsf{distl}) *^{\|} \cdot \mathsf{distl}^{\|} \cdot (\mathsf{pf} \cdot \pi_1, \pi_2)^{\circ})^{\circ} \cdot \\ \quad (\mathsf{split}_p \cdot \pi_1, \mathsf{split}_q \cdot \pi_2)^{\circ}\end{array} \qquad (52)$$

Figure 136. Paralleliser rule for handling select functions

initial implementation uses the first rule (rule 51). Later refinements will use the second.

Select select takes a pair consisting of a source vector s and a target vector of integers: $[t_0, \ldots, t_{n-1}]$ and produces a result vector, $[s!t_0, \ldots, s!t_{n-1}]$ consisting of the elements of the source vector indexed by the target vector. A simple rule for the parallelisation of select is:

$$\mathsf{split}_p \cdot \mathsf{select} \Rightarrow (\mathsf{select}) *^{\|} \cdot \mathsf{distl} \cdot (\pi_1, \mathsf{split}_p \cdot \pi_2)^{\circ}$$

which produces code that sends the entire source vector for select to each node for selection according to the distributed target vector. This code is not scalable to many nodes, so a rule is needed that distributes only the required source data to each node.

A better rule is shown in figure 136. The new distselect (distributed-select) function takes a pair of distributed vectors of the form:

$$(s, t)$$

where:

$$s = [[s_{(0,0)}, \ldots, s_{(0,n_0-1)}], [s_{(1,0)}, \ldots, s_{(1,n_1-1)}], \ldots, [s_{(0,m)}, \ldots, s_{(0,n_{m-1}-1)}]]$$

and where:

$$t = [[t_{(0,0)}, \ldots, t_{(0,k_0-1)}], [t_{(1,0)}, \ldots, t_{(1,k_1-1)}], \ldots, [t_{(0,m)}, \ldots, t_{(0,k_{l-1}-1)}]]$$

Given this pair, distselect returns a distributed vector:

$$[[s!t_{(0,0)}, \ldots, s!t_{(0,k_0-1)}], [s!t_{(1,0)}, \ldots, s!t_{(1,k_1-1)}], \ldots [s!t_{(0,m)}, \ldots, s!t_{(0,k_{l-1}-1)}]]$$

Note that each $t_{(i,j)}$ is a pair of values and each application $s!t_{(i,j)}$ of an indexing function above, is implicitly nested.

distselect works in two phases. The first phase requests the values indexed by the target vector. The second phase sends the requested values to their destinations. Figure 137 shows the two phases of the execution of:

$$\begin{aligned}\mathsf{distselect}\ (\ &[[a,b,c],[d,e],[f,g],[h,i]],\\ &[[(0,2),(3,1),(0,2)],[(1,1),(2,0),(2,1)],[(0,1),(0,0)]])\end{aligned}$$

which generates the distributed vector $[[c,i,c],[e,f,g],[b,a]]$.

The parallelised code upstream of distselect in figure 136:

$$(\pi_1, (\mathsf{app}) **^{\parallel} \cdot (\mathsf{distl}) *^{\parallel} \cdot \mathsf{distl}^{\parallel} \cdot (\mathsf{pf} \cdot \pi_1, \pi_2)^{\circ})^{\circ} \cdot (\mathsf{split}_p \cdot \pi_1 \mathsf{split}_q \cdot \pi_2)^{\circ}$$

is responsible for distributing the source and target vector using:

$$(\mathsf{split}_p \cdot \pi_1, \mathsf{split}_q \cdot \pi_2)^{\circ}$$

extracting the closure of the partitioning function of the source vector using:

$$\mathsf{pf} \cdot \pi_1$$

Distributing this closure of the partitioning function over the target array using:

$$(\mathsf{distl}) *^{\parallel} \cdot \mathsf{distl}^{\parallel}$$

and, finally, applying the partitioning function to each index of the target vector to produce the required distributed vector of pairs using:

$$(\mathsf{app}) **^{\parallel}$$

distselect embeds a non-uniform, personalised, many-to-many communications pattern[90]. This is the most general of communications patterns, and is not simple to implement efficiently. We have found it simpler, in an initial implementation, to embed distselect in an all-to-all or total exchange primitive of the sort found in BSP[76][18].

[18]BSPLib[75] provides much of the functionality required by distselect through its direct memory access primitive: `get`.

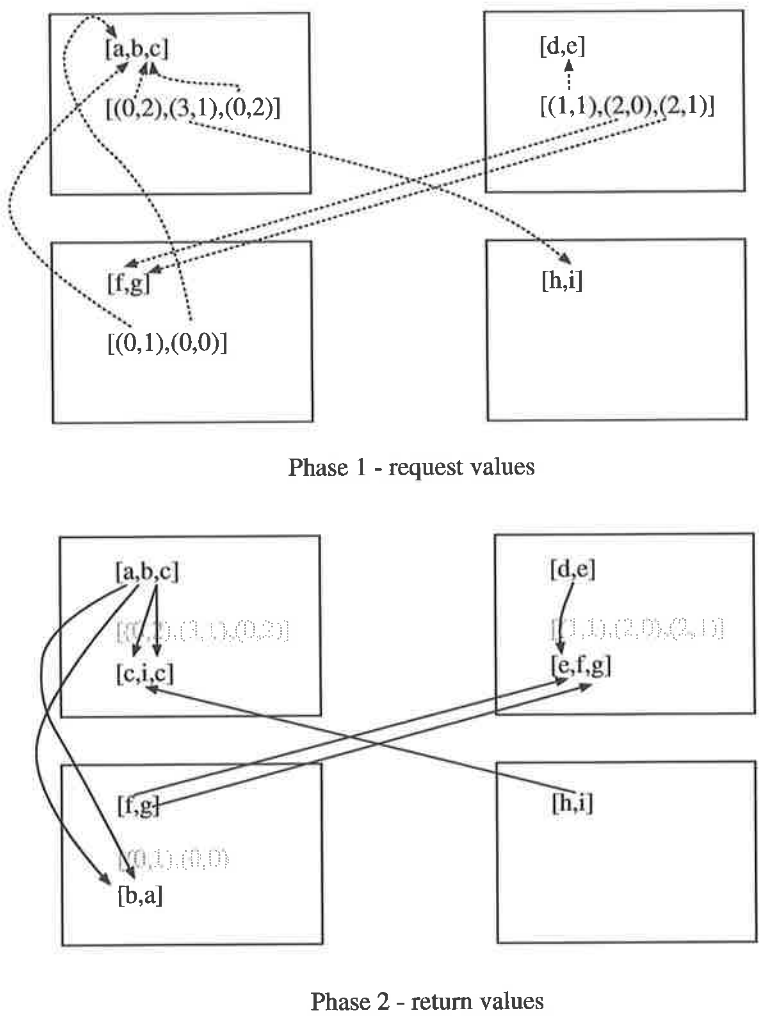


Figure 137. The operation of distselect. In the first phase requests are generated from the target vector and sent, along with a return address. In the second phase, copies of the requested values are copied from the source vector, generating a new array distributed in the same way as the target vector.

If functions

$$\frac{\begin{array}{c}\mathsf{id}_{\alpha\to\mathtt{boolean}} \cdot \mathit{pred} \Rightarrow \mathit{pred}' \cdot s' \\ s \cdot \mathit{consq} \Rightarrow \mathit{consq} \cdot s'' \\ s \cdot \mathit{alt} \Rightarrow \mathit{alt}' \cdot s'''\end{array}}{s \cdot \mathsf{if}(\mathit{pred}, \mathit{consq}, \mathit{alt}) \Rightarrow \mathsf{if}(\mathit{pred}' \cdot^{3} \pi_1, \mathit{consq} \cdot^{3} \pi_2, \mathit{alt} \cdot^{3} \pi_3) \cdot (s', s'', s''')^{\circ}} \quad (53)$$

Figure 138. Paralleliser rule for handling if functions

Specialisations of select Where there is some static knowledge of the pattern of communication created by an instance of select. More specialised primitives can be used in its place[19]. For example:

$$\mathsf{select} \cdot (\mathsf{id}, (\mathsf{mod} \cdot (+ \cdot (\pi_2, 1)^{\circ}, \pi_1)^{\circ}) * \mathsf{distl} \cdot (\#, \mathsf{iota} \cdot \#)^{\circ})^{\circ}$$

is the same as:

$$\mathsf{cshift} \cdot (\mathsf{id}, 1)^{\circ}$$

irrespective of the input value of this code. The use of such specialisations is future work[20].

If functions Figure 138 shows the parallelisation rule for if functions. The premises of this rule parallelise the predicate (*pred*), consequent (*consq*), and alternative (*alt*) functions of the if function. Note that *pred* is parallelised with respect to an id distributor function rather than with respect to s because the boolean output of *pred* is consumed by the if function rather than functions downstream. *consq* and *alt* are parallelised with respect to s.

The conclusion of the rule embeds all three distributor functions generated by the premises into an alltup function, $(s', s'', s''')^{\circ}$ and inserts addressing functions upstream of each of *pred*$'$, *consq*$'$, and *alt*.

Note, that this rule is designed to cater for the worst-case scenario where the new distributor functions, s', s'' and s''' are all distinct. This can happen, for example, when each component of the if function accesses different parts of an input tuple.

[19]There is much work on establishing and exploiting statically knowable patterns of reference. Von Hanxleden et al. [141] recognise varying levels of static knowledge in code in their work aimed at identifying and aggregating overlapping communications. The worst case code allows analysis at the *portion-level* and the best case allows analysis at the *reference-level*.

[20]Such specialisations are, essentially, *strength-reductions* on the distselect operator.

While functions

$$\frac{\begin{array}{c} s_{i\to o}\cdot f \Rightarrow f'_{i'\to o'}\cdot s' \\ \mathsf{id}\cdot p \Rightarrow p'_{i''\to \texttt{boolean}}\cdot s'' \\ \mathit{Reconcile\_types}(i', o \Rightarrow r) \\ \mathit{Reconcile\_types}(i'', o \Rightarrow r') \end{array}}{s_{i\to o}\cdot \mathsf{while}(f,p) \Rightarrow \mathsf{while}(f'\cdot r, p'\cdot r')\cdot s_{i\to o}} \quad (54)$$

Figure 139. Paralleliser rule for handling while functions

If there is any overlap between s', s'' and s''' then there is scope for introducing substantial efficiency gain through common-sub-expression elimination[21].

While With the while function, the output value of one iteration is usually the input value to the next iteration. As a consequence, the input and output types of a while must be the same and, moreover, the parallelisation process must preserve this property by ensuring that data that is distributed in the input to the while function is also distributed in its output.

Figure 139 shows the parallelisation rule for while functions. The basic strategy of the rule is:

1. Move the distributor function s from the downstream side of the while function to the upstream side.
2. parallelise the code in the loop-body f and the predicate function p to compensate.

The first step in the strategy is trivial and is done in the conclusion of the rule. The second step is non-trivial and is carried out in the premises of the rule.

The first premise parallelises f with respect to the distributor function s producing f'. The second premise parallelises the predicate function p with respect to id[22]. Note that there is the potential for a mis-match in the distribution of output produced by s and the distribution of input expected by f' and p'. To be more precise, s may

[21] Alternatively, we could, at little cost, introduce more rules for to capture the special cases where two or more of the new distributor functions are the same.

[22] As with the predicate in the if function, the id function is needed because all of the boolean output is consumed, internally, by the while function.

Set *Reconcile_Types* **is** $\boxed{TYPE, TYPE \Rightarrow B\_EXP}$

Identical types

$$\alpha, \alpha \Rightarrow \mathsf{id}_{\alpha\rightarrow\alpha} \tag{55}$$

Distributed input expected, non distributed given

$$[[\alpha]], [\alpha] \Rightarrow \mathsf{split}_{p\,[\alpha]\rightarrow[[\alpha]]} \tag{56}$$

Non-distributed input expected, distributed given

$$[\alpha], [[\alpha]] \Rightarrow +\!\!+ /_{[[\alpha]]\rightarrow[\alpha]} \tag{57}$$

Alltup types

$$\frac{\alpha_1, \beta_1 \Rightarrow f_1 \quad \ldots \quad \alpha_n, \beta_n \Rightarrow f_n}{(\alpha_1, \ldots, \alpha_n), (\beta_1, \ldots, \beta_n) \Rightarrow (f_1 \cdot \pi_1, \ldots, f_n \cdot \pi_n)^\circ} \tag{58}$$

End *Reconcile_Types*

Figure 140. Rules for generating a type reconciliation function

produce a value containing a distributed vector where f' and/or p' expects a non-distributed vector, or vice-versa. This potential conflict is circumvented by the calls to *Reconcile_types* in the last two premises. These calls compare the distribution of vector values in the input types of f' and p' with that of corresponding vectors in the output type of s, and interpose the appropriate distributor/gatherer function if required[23]. Figure 140 shows the rules of *Reconcile_types*. Rule 55 applies when the two types are identical. No transformation is needed to reconcile the types so an id function is produced.

Rule 56 applies when distributed input is expected but the distributor function produces a non-distributed value. In this case, a split_p function is inserted to deliver the distributed value.

Rule 57 applies when non-distributed input is expected but the distributor function produces a distributed value. In this case, a $+\!\!+/$ function is produced

[23]Because the current set of rules aggressively parallelises most vector values the potential for a clash between distributed and non-distributed vector values is small. However, we envisage that in future implementations parallelisation will not be so aggressively pursued and the potential for conflict will thus increase.

Transpose functions

$$\frac{a > 0 \qquad b > 0}{\mathsf{split}_p \cdot \mathsf{transpose}_{a,b} \Rightarrow +\!\!+ /^{\|} \cdot ((\mathsf{transpose}_{a-1,b-1})*) *^{\|} \cdot \mathsf{split}_p} \qquad (59)$$

Figure 141. Paralleliser rule for handling **transpose** functions on the non-outer dimensions of vectors.

Default rule

$$\frac{\mathit{Gen\_Suffix}(\alpha \Rightarrow s' \cdot s'')}{s \cdot f_{\beta\to\alpha} \Rightarrow s \cdot f_{\beta\to\alpha} \cdot s' \cdot s''} \qquad (60)$$

End *Parallelise*

Figure 142. Default parallelisation rule

to flatten the data.

Finally, rule 58 applies this rule-set recursively in tuples.

transpose Figure 141 shows a rule for **transpose** on non-outer dimensions of a nested vector. A rule for instances of **transpose** involving the outer dimension of nested vectors is future work.

Default rule The final rule of *Parallelise* is shown in figure 142. This rule captures any code not caught by the preceding rules. It simply leaves the old distributor function s in place and builds a new suffix function $s' \cdot s''$ upstream of f for further processing.

This rule works well for scalar functions, including most binary and unary operators[24]. However, this rule also currently applies to functions such as **iota** and **priffle** where there is still potential parallelism to be exploited. Devising specific rules for other, potentially, parallel functions remains future work.

This concludes our description of the parallelisation process. The rules here were applied, by hand[25], to generate the parallel BMF code directly executed by

[24]The rule, temporarily, introduces redundant identity functions into programs but these are easily eliminated by a final call to the *Remove_ids* rule set.

[25]A prototype automated paralleliser based, in part, on an earlier version of these rules is described in [146].

the simulator, described in section 6.4.2, in the experiments in section 6.4.3.

In order for these experiments to provide realistic results it is important to, first, consider how the code generated from parallelised BMF will perform on a target architecture. We briefly explore the performance of generated code next.

6.3 Code generation

The parallelisation process produces BMF code suitable for targeting to an abstract architecture. The job of code generation is to map this code onto a concrete architecture.

6.3.1 Preliminary implementation

A preliminary implementation of a code generator from parallel BMF to C with calls to MPI has been constructed by Dean Philp [111]. This implementation used a reasonably direct mapping of BMF constructs to C-MPI.

Code for each BMF construct is generated using simple rules that map each function to a block of C code. The monomorphic type specifications that accompany each BMF function are used to guide storage allocation, message construction and message decomposition. Minor issues arise from the need to separate declarations and statements in some versions of C but these are easily dealt with.

The code generation phase can be divided into two major aspects. The first is the generation of code for distributed parallel functions and the second is the generation of code for the sequential, on-node, functions.

6.3.1.1 Distributed functions

Distributed functions are mapped to their closest analogue in MPI[26]. For example a map is converted to a simple parallel code-block. Function composition between

[26]Early experiments with the platform used by our implementation[101] indicate that in, at least in some cases, there is little to be gained by writing custom operators in terms of send and receive. Much, depends of course, on the underlying MPI implementation. Grove [66] has done much work in sampling the behaviour of various constructs in various implementations. The quality of these vary greatly. It is also possible, once we have established what the precise needs of our implementation are, to write more specialised and efficient library functions than the robust, general-purpose calls offered by MPI which may also make a sensible platform for our implementation.

distributed functions is converted to a barrier-synchronisation[27]. reduce is converted to MPI_Reduce and scan is converted to MPI_Scan.

Messages are sent as generic blocks of data and efficient code is produced to compose and decompose messages. The allocation of space for incoming and outgoing messages is handled on as-needed basis with deallocation occurring as soon as the need for the storage lapses. This memory management scheme, though not optimal, is a reasonable one for message passing operations, which are heavyweight and expensive enough to dominate their message allocation/deallocation costs.

6.3.1.2 On-node functions

Functions that execute on a single node are converted, using mostly simple rules, directly to C code. Most vector operations such as map, reduce, scan, iota, select, distl, distl, zip are represented as a loop.

In the current implementation, space is allocated for every function's output value prior to its execution and the space allocated to its input is deallocated after its execution. Addressing functions are converted into references into this allocated space, cast to the appropriate type. This pattern of frequent allocation and deallocation leads to high memory management costs. We explore this problem, and how it might be ameliorated, next.

6.3.2 Memory Management Costs

The memory-management scheme, described above, has the advantage of being simple and robust, with straightforward semantics and a simple garbage-collection scheme. However, the frequent calls to `malloc` and `free` that this scheme generates will incur very significant run-time overheads.

To help quantify these overheads we ran a series of small experiments, described in Appendix F, to estimate the average costs of allocation and deallocation relative to the cost of executing a single instruction. From these experiments we derived estimates of 100 instructions for each call to `malloc` and 70 instructions for each call to `free`. Details are covered in the Appendix but it is worth noting that there is much variation in these costs, with some calls taking much longer. In any case, the

[27]The cost of barrier-synchronisation is a good reason for fusing distributed operations[62]. All other things being equal, the fewer barrier synchronisations the better. This type of optimisation is a strong candidate for inclusion in future versions of the Adl implementation.

costs involved with each call are such that they can significantly diminish performance unless properly managed.

It should also be noted that, along with the raw cost of calls to `malloc` and `free` our simple memory management scheme also further increases overhead in copying structures into newly allocated memory, especially when large data structures are involved.

Similarly large memory management costs have been encountered in other domains. Perhaps the most relevant example comes from performance figures collected from executing APL interpreters in Bernecky's description of the APEX compiler [16].

The costs of memory management are typically near 20% of the total cost of program execution in an *interpreter*. If we take this cost and transfer it into an environment where a program is *compiled*, where we can expect other overheads to be much smaller, this relative cost would rise dramatically. The interpreters tested by Bernecky use a very similar memory management scheme to ours[28]. Clearly, this is an issue that needs to be addressed in future versions of our implementation.

6.3.2.1 Overcoming Memory Management Costs

High on-node memory management costs come from a strict adherence to the functional semantics of BMF where values are not updated but created anew. These costs are compounded in point-free form where no value can survive outside of a function. Essentially, there is a mis-match between the programming model which eschews the recycling of variables using update and the architectural model which thrives on such recycling. Code generation must bridge this gap in the most efficient way possible.

Fortunately, there is successful work to draw on in this area. The SISAL compiler [55, 56] uses copy-elimination, reference count elimination and other forms of update-in-place analysis to convert IF1, a functional form, into IF2, an imperative form. These transformations are crucial to the efficiency of the, very effective, Optimising SISAL Compiler (OSC).

Bernecky's APL compiler, mentioned earlier, produced SISAL target code partly in an attempt to leverage the successful memory management of OSC. Interestingly,

[28]Bernecky, reduced these costs, in part, by exploiting the Optimising SISAL Compiler, which has sophisticated strategies for reducing memory management costs.

it was found that a mismatch in the primitive types of SISAL and APL meant that substantial analysis had to be carried out by the APL compiler as well.

Walinsky and Banerjee's FP* compiler used a routing-function elimination phase to generate efficient imperative code from an FP* program. All of these implementations have made considerable gains in efficiency.

It must be noted that memory management issues have not been solved in general for functional implementations and applications[72]. However, even with these constraints in mind, there is reason to be optimistic of the prospect of very significant improvements in memory management costs in our implementation before problems needing a more advanced treatment become an issue. In short, the problem of memory management in functional languages is non-trivial but work with SISAL and APL has shown it can be tackled effectively for a large range of programs.

This completes the summary of the nascent code-generation process. We use the information we have gathered from these investigations to form a realistic basis for our simulator. Next we step back to review the performance of parallelised BMF code on a simulated architecture.

6.4 Results

Thus far in this chapter we have described preliminary work on parallelisation and code generation. In this section we use a detailed simulator to gain some insight into how effective the parallelisation process is. This section:

- describes the experimental methodology including a brief description of the simulator
- compares the performance of parallelised translator and optimiser code;
- characterises the impact of different memory-management costs;
- demonstrates that this process generates speedup and points to where it can be found.

The results of the simulation are accompanied by static analysis where appropriate. We describe the simulation process next.

6.4.1 Methodology

Our results for each program are obtained through the following process.

- Manual parallelisation of code, followed by
- Execution of the parallelised code on a detailed simulator.

The manual parallelisation step is carried out according to the parallelisation rules from the earlier part of this chapter. The simulator warrants a more detailed discussion at this point.

6.4.2 The simulator

The simulator is based on a detailed model of a distributed architecture built by Paul Martinaitis[99]. The following parameters are configurable:

- The number of nodes.
- The topology of the interconnect linking nodes.
- The latency and bandwidth associated with each link in the interconnect.
- Message startup costs.
- The on-node cost of each primitive instruction.

The simulator interprets parallel and sequential BMF instructions directly, mapping them onto an underlying virtual distributed-memory machine. The costs of allocation and deallocation form part of the total cost of executing each instruction. Routing of messages between nodes is statically determined according to minimum cost functions. The parameter values used for the experiments in this chapter are:

Interconnect: A 32-node cross-bar, and in one case, a 8x4 store-and-forward mesh[29].

Latency: $5\mu s$ ($5000ns$) per-link including message startup costs[30]

[29]We used two configurations test the simulator on more than one type of network.

[30]This is figure is quite low but some interconnects such as Quadrics' QS-1[96] better it, and Myrinet[83] approaches it.

Bandwidth: a peak bandwidth of $3ns/byte$ is assumed. The effective bandwidth measured by the simulator can, of course, be much worse when messages are small. We assume that the maximum packet-size for the interconnect is arbitrarily large, so effective bandwidth asymptotically approaches its maximum as message size increases.

Instruction costs: one time unit for each primitive instruction plus allocation, deallocation and copying costs. We have high and low settings for memory management costs. Under the high setting we have a cost of 100 time units for allocation, irrespective of block-size and 70 time units for deallocation. Under the low setting we have a cost of two time units for allocation and one time unit for deallocation. The high cost setting simulates the performance of code with no optimisation of memory management costs, while the low setting simulates the performance after optimisation of memory costs[31].

The cost of copying data remains the same under both the high and low cost models. When a distributed vector is copied, it is, reasonably, assumed that the copying of elements takes place in parallel. Ideally, the low-cost model would minimise copying costs as well as allocation and deallocation costs but such minimisation requires further analysis to simulate in a sensible manner[32].

The simulator also provides a detailed execution and communications trace suitable for display in Paragraph[73] and is also able, to a limited extent, simulate congestion on network links. We see some traces later in this chapter. The ability to simulate congestion is not exploited here.

6.4.3 Experiments

We describe the outcomes of experiments on parallelised BMF code produced by the following programs:

[31]The low-cost memory management setting is a compromise. The substantial costs of copying data are still counted under the low-cost model. However this is somewhat compensated for by the assumption that *all* allocation costs are reduced under the low-cost model which is, a-priori, somewhat optimistic.

[32]It should be noted that the simulator is written to avoid a reasonable amount of gratuitous copying. The copying that does remain will probably require some contextual analysis for it to be reduced.

`map_map_addconst.Adl` a simple nested map, parallelised in the outer dimension. We also run a small experiment with `map_map_atan.Adl` which has identical structure but a more expensive operator.

`sum.Adl` a simple reduction. We also run an experiment with `mss_reduce.Adl`, a reduction with a moderately costly binary operator.

`mss.Adl` equivalent to `mss_reduce.Adl` but performing a parallel scan instead of reduce.

`finite_diff.Adl` a one-dimensional stencil algorithm containing a `while` loop.

`remote.Adl` a simple convolution.

For each of these programs we compare the performance of translator and optimiser code under both high-cost and low-cost memory management regimes. The data-sizes used for experiments involving translator code are, by necessity, small due to the large data structures created by translator code and some limitations of the simulator. Additional experiments, involving only optimised code, are run, under both high and low-cost memory management regimes, often for a variety of, somewhat larger, data-sizes. Static analysis is provided where it helps the interpretation of the results.

It should be noted that, in terms of performance, experiments fall into three groups.

1. `map_map_addconst.Adl` and `sum.Adl`: these applications are very simple and have low computation to communication ratios for all sizes of input data. These applications are not expected to generate speedup on most distributed memory machines. These examples help validate the cost model and help set up the descriptive context.

2. `map_map_atan.Adl`, `mss_reduce.Adl` and `mss.Adl`: These applications have higher computation to communication ratios that stay relatively constant as input data grows. These applications will generate modest speedup on fast distributed memory machines. Interestingly, if we extrapolate the performance of processors and networks it can be expected that this modest speedup will disappear over time as computing power continues to increase relative to network bandwith.

$$+\!\!+ /^{||} \cdot ((+ \cdot (\pi_2, 2)^\circ) * \cdot \mathsf{distl} \cdot (\mathsf{id}, \pi_2)^\circ \cdot \mathsf{id}) * *^{||} \cdot (\mathsf{distl}) *^{||} \cdot \mathsf{distl}^{||} \cdot (\mathsf{id}, \mathsf{split}_p \cdot \mathsf{id})^\circ \cdot \mathsf{id}$$

(a)

$$+\!\!+ /^{||} \cdot (+ \cdot (\mathsf{id}, 2)^\circ) * * *^{||} \cdot \mathsf{split}_p$$

(b)

Figure 143. Parallelised translator code (part (a)) and optimiser code (part (b)) for `map_map_addconst.Adl`

3. `finite_diff.Adl` and `remote.Adl`: these applications have a computation to communication ratio that grows as input data size grows. Given enough data, these applications can generate substantial speedup on most distributed platforms.

Measures for speedup are made relative to the speed of the same application running on a single processor. So, for example, to calculate the speedup of parallelised translator code running on a number of processors it is compared to the speed of that same parallelised translator code running on a single processor[33]. Likewise, parallelised optimised code is compared with itself on different numbers of processors.

The experiments are presented in rough order of complexity starting with: `map_map_addconst.Adl`.

6.4.4 Experiment 1: `map_map_addconst.Adl`

This program adds a constant to each element of a nested input vector. The structure of the program is a simple, nested map.

Figure 143 shows the parallelised translator and optimiser code for `map_map_addconst.Adl`. The parallelised translator code splits the outer dimension of one copy of the input vector into p segments, then distributes copies of the input vector over those segments and then performs a parallel map over those segments. The last stage concatenates the distributed vector back on to a single node.

[33]We use this measure for convenience. A slightly more accurate measure is obtained by comparing the speed of the best sequential program against the parallel versions of the code. However, in the experiments presented in this chapter, the difference between this ideal measure and the measure we actually use is small.

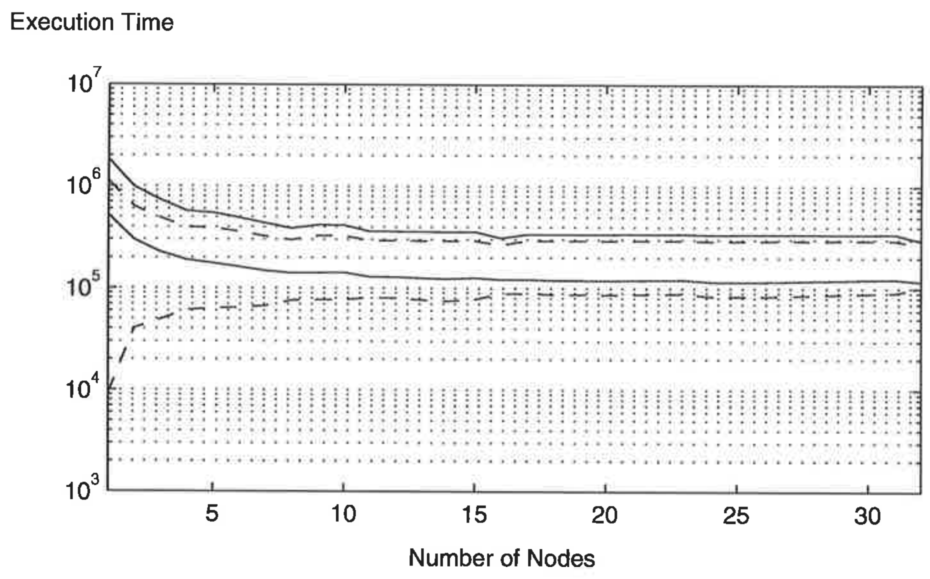


Figure 144. Run-times of parallel translator and optimiser code for `map_map_addconst.Adl` on between 1 and 32 nodes of a crossbar-connected machine with 32x32 input values under a high-cost regime (solid lines) and low cost regime (dash lines).

The parallelised optimiser code splits the outer dimension of the vector into p segments, performs a parallel map over those segments and then concatenates the distributed vector back on to a single node.

In our first experiment, translator and optimiser code were run against a nested 32 by 32 value input vector utilising between 1 to 32 nodes under the high-cost and low-cost memory management regimes. Figure 144 shows the run-times of translator and optimiser code, on a logarithmic scale, under these regimes. Times for the high-cost regime are plotted with a solid line and times for the low-cost regime are plotted using dashed lines. At all points, optimiser code is several times faster than translator code, due to the larger data structures allocated by the translator code and the consequent costs of transmitting these structures in the distributed case. Moreover, the sizes of intermediate structures in unoptimised code grow faster with respect to input data size than corresponding structures in optimised code. That is, we can expect this gap to grow with the size of input data.

Predictably, there is also a significant performance gap between programs running under the high-cost and low-cost memory allocation regimes. This gap closes as the number of nodes increases and communication costs start to dominate over memory allocation costs. Most interestingly, the optimised code under the high-cost regime

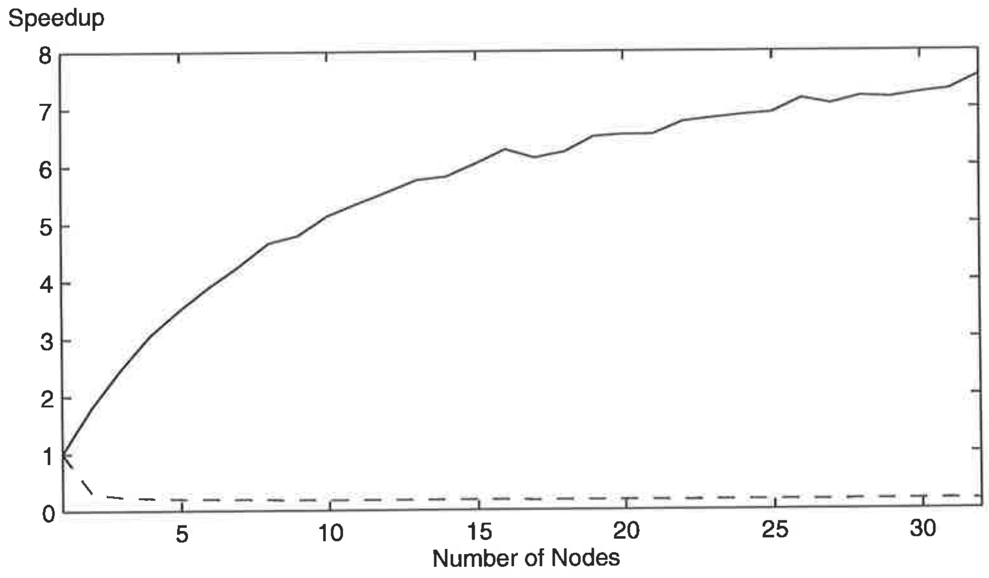


Figure 145. Speedup for optimised code for `map_map_addconst.Adl` applied to a 128 by 128 element input vector on between 1 and 32 nodes of a crossbar-connected machine under both high-cost (solid line) and low-cost (dashed-line) memory management regimes.

gives some speedup and the optimised code under the low-cost regime does not.

At it turns out, this is due to the poor ratio of computation to communication in `map_map_addconst.Adl`. To verify this, when we increase the amount of computation per-node by increasing the input data size to the optimised program by a factor of 16 we get the performance profile shown in figure 145. Even with this larger input data there is still no evidence of speedup under the low-cost regime. Closer analysis of the costs of the parts of the optimised code reveals why this is so. The computation done by the body of the innermost map:

$$+ \cdot (\mathsf{id}, 2)^\circ$$

takes $8ns$ to run. This code is fuelled by one word of data that, solely in terms of bandwidth, takes $24ns$ to transmit. With such a gap between marginal computation and communication costs, increasing the total amount of computation per-node will result in a loss rather than a gain in performance due to parallelism[34].

[34]Note that such bandwidth bottlenecks are well understood and have also been well studied in a single-processor context. Using the terminology of Ding and Kennedy [50] the $8ns$(8 instruction) figure represents *program balance* and the $24ns$(24 instruction) figure represents *machine balance*. In a distributed context, the difference is the minimum marginal loss from distributing that computation

$$+\!\!+ /^{\parallel} \cdot (\mathsf{atan}) * * *^{\parallel} \cdot \mathsf{split}_p$$

Figure 146. Optimised code for `map_map_addconst.Adl` with atan substituted for the original map body.

The unoptimised program, though less efficient in absolute terms, carries out a lot more (redundant) computation per-word of transmitted data and, thus, can deliver quite respectable speedup. Likewise, the high memory management cost regime can also, to a large extent, hide the imbalance. This slightly perverse outcome is well understood, as noted by Rüde[122]:

> For successful parallel computing it is helpful to parallelise the worst sequential program one can find.

Speedup can only be attained for the optimised version of `map_map_addconst.Adl` by modifying our estimates of architectural parameters to increase either bandwidth or average instruction costs [35].

If speedup is not easily attainable, under the low-cost regime, for the optimised `map_map_addconst.Adl` could speedup be expected from a similar program? The answer is yes. If we modify the optimised code so that instead of an addition operation the inner loop contains an approximating function such as atan to produce the code in figure 146 we can attain some speedup as shown in figure 147. In this experiment we assume that the cost of an atan operation is $64ns$. This cost estimate, which will vary from machine to machine, is comparable to the cost for atan on the Itanium processor (circa 2001[65]).

Only a small amount of speedup is manifest in figure 147. As with the original `map_map_addconst.Adl`, bandwidth is the bottleneck. To see why this is the case we generalise for all programs whose basic structure, like that of `map_map_addconst.Adl`, consists of a single split followed by a fixed-sized sequential computation on each node, followed by a reduce with concatenate.

($16ns$). In both a single-processor and distributed context, careful program design can ameliorate but not eliminate the problem.

[35] Architectures with greater bandwidths than our model do exist. For example, the IBM Regatta-H cluster has a bandwidth of over *2GB/Sec* [67] between its closely connected nodes which translates into a marginal cost of less than *0.5ns/byte*, over six times better than our model. Unfortunately, such good performance is relatively rare and expensive. Increasing estimates of average instruction costs also seems unrealistic, though we haven't considered any cache effects that might increase average instruction costs as the problem size increases.

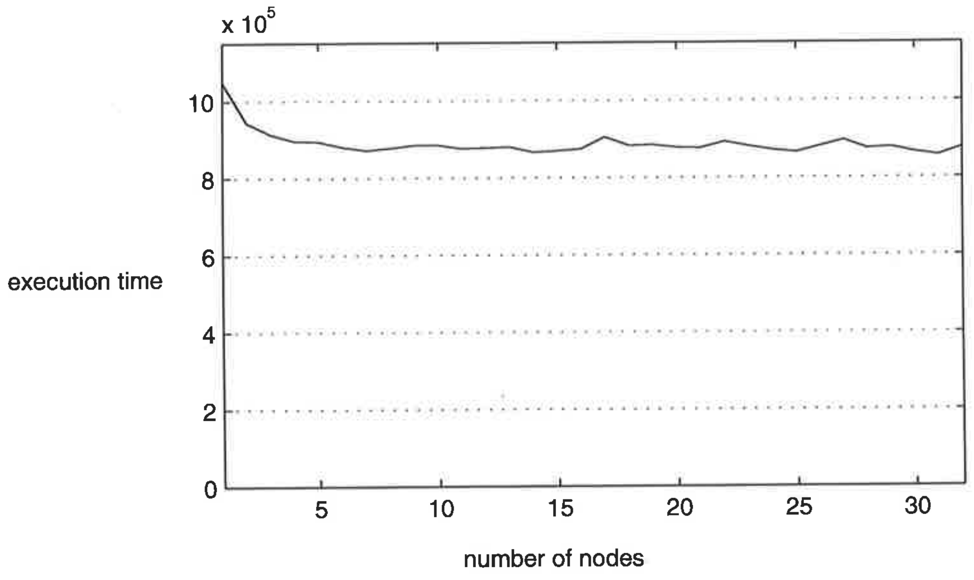


Figure 147. Execution time for program from figure 146 on between 1 and 32 crossbar-connected nodes on a 128x128 element input vector. The cost of an `atan` operation is assumed to be $64ns$.

The relative execution time of any parallel code: T_p relative to its sequential counterpart: T_1 is $\frac{T_p}{T_1}$. For any parallel application where communication and computation are not allowed to overlap the following (specialised from[10]) holds:

$$\frac{T_p}{T_1} = A + \frac{(1-A)}{p} + \frac{t_{comms}}{t_{comp}} \tag{61}$$

Where:

- A is the Amdahl fraction of the program, the inherently sequential portion,
- p is the number of processors,
- t_{comms} is the time taken to perform communications.
- t_{comp} is the time taken to perform computations. It is the time taken to run the parallel program on one processor. Most work, including this work, assumes $t_{comp} = T_1$.

In many parallel applications A is either small or shrinks rapidly as the problem size grows. For these applications, which includes all of those in the experiments in this chapter, the following approximation serves well:

$$\frac{T_p}{T_1} \approx \frac{1}{p} + \frac{t_{comms}}{t_{comp}} \tag{62}$$

t_{comp} for the program in figure 146 is defined:

$$t_{comp} = t_{instr} c_{atan} n \tag{63}$$

where t_{instr} is the time it takes to execute one primitive instruction, c_{atan} is the number of instructions executed by **atan** (64) and n is the number of elements of input data. In the previous experiment we assumed amounts of $1ns$, 64 and $128 \times 128 = 16384$ for these, respectively. t_{comms} is defined:

$$t_{comms} = t_{latency} + t_{bandwidth} \tag{64}$$

where, for the program in figure 146, running on a crossbar,

$$t_{latency} = (2 \log p) t_{link\_latency} \tag{65}$$

and

$$t_{bandwidth} = (2 - \frac{2}{p}) n \beta \tag{66}$$

where $t_{link\_latency}$ is the link latency ($5000ns$ in our model) and β is the amount of time it takes to transmit an extra word on a link ($24ns$ in our model).

If we substitute eqns (64,63,65, 66) into eqn (62) we get:

$$\frac{T_p}{T_1} \approx \frac{1}{p} + \frac{(2 \log p) t_{link\_latency} + (2 - \frac{2}{p}) n \beta}{t_{instr} c_{atan} n} \tag{67}$$

As both p and n grow the terms $\frac{1}{p}$ and $(2 \log p) t_{link\_latency}$ become insignificant, and $(2 - \frac{2}{p})$ approaches 2, leaving:

$$\frac{T_p}{T_1} \approx \frac{2n\beta}{t_{instr} c_{atan} n} = \frac{2\beta}{t_{instr} c_{atan}}$$

which in our experiment is: 48/64 giving a maximum speedup of: 1.33 which is in line with the results in figure 147. In both cases the amount of speedup extracted is small and asymptotically limited to a constant. It should be noted that this small speedup is not an inherent property of parallel computation using **map**. If the amount of computation inside the **map** body can be scaled up to swamp the initial costs of distribution the benefits of parallelism can be substantial. This point is illustrated soon in the discussion of the performance of `finite_diff.Adl`.

$$
\begin{array}{l}
\mathsf{if}(\ \neq \cdot(0, \# \cdot \pi_2)^\circ, \\
\quad \pi_2 \cdot ((\pi_1 \cdot \pi_1, + \cdot (\pi_1 \cdot \pi_1, (\pi_2 \cdot \pi_1, \pi_2 \cdot \pi_2)^\circ)^\circ)^\circ)/^{||} \cdot \\
\quad (\cdot((\pi_1 \cdot \pi_1, + \cdot (\pi_1 \cdot \pi_1, (\pi_2 \cdot \pi_1, \pi_2 \cdot \pi_2)^\circ)^\circ)^\circ)/) *^{||} \cdot(\mathsf{distl}) *^{||} \cdot \mathsf{distl}^{||}, \\
\quad \mathsf{0} \cdot \pi_1) \\
\cdot(\mathsf{id}, \mathsf{split}_p \cdot \mathsf{id})^\circ
\end{array}
$$

(a)

$$+/_0^{||} \cdot (+/_0) *^{||} \cdot \mathsf{split}_p$$

(b)

Figure 148. Parallelised translator code (a) and optimiser-code (b) for `sum.Adl`

6.4.5 Experiment 2: simple reductions

Chapter 5 used the program `sum.Adl` to show the effect of the optimiser on simple reductions. The parallelised translator and optimiser code for `sum.Adl` is shown in figure 148. The execution time of these programs when applied to 512 data items for different numbers of processors is shown in figure 149. As with `map_map_addconst.Adl` the performance of optimised code is substantially better than that of non-optimised code under both the high and low-cost memory management regimes. Again, there is better absolute performance under the low-cost memory allocation regime (the dashed line) but the speedup under this regime for optimised code is poor.

This poor speedup persists as input data size increases. Figure 150 shows the speedup of optimised code for `sum.Adl` under both the high-cost (solid line) and low-cost (dashed line) memory management regimes for a input vector with 4096 values. Some speedup is evident under the high-cost memory management regime but there is very poor speedup under the low-cost regime.

Again, this lack of speedup is due to program balance. The equation for the runtime on p nodes relative to the runtime on one node for parallelised programs with the structure of `sum.Adl` is:

$$\frac{T_p}{T_1} \approx \frac{1}{p} + \frac{(2\log p)t_{link\_latency} + ((1-\frac{1}{p})n\beta + (1-\frac{1}{p})\beta}{t_{instr}c_{plus}(n-1)} \tag{68}$$

as n and p become large the $\frac{1}{p}$,$(2\log p)t_{link\_latency}$, and $(1-\frac{1}{p})\beta$ terms become less

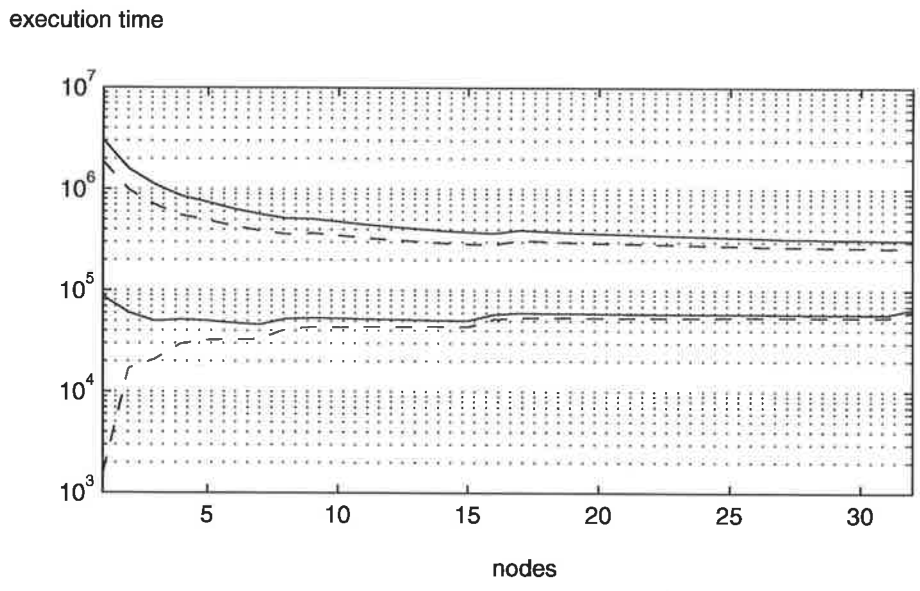


Figure 149. Relative performance of translator versus optimiser code for `sum.Adl` applied to 512 input values on 1 to 32 crossbar-connected nodes under the high-cost (solid-line) and low-cost (dashed-line) model of memory management.

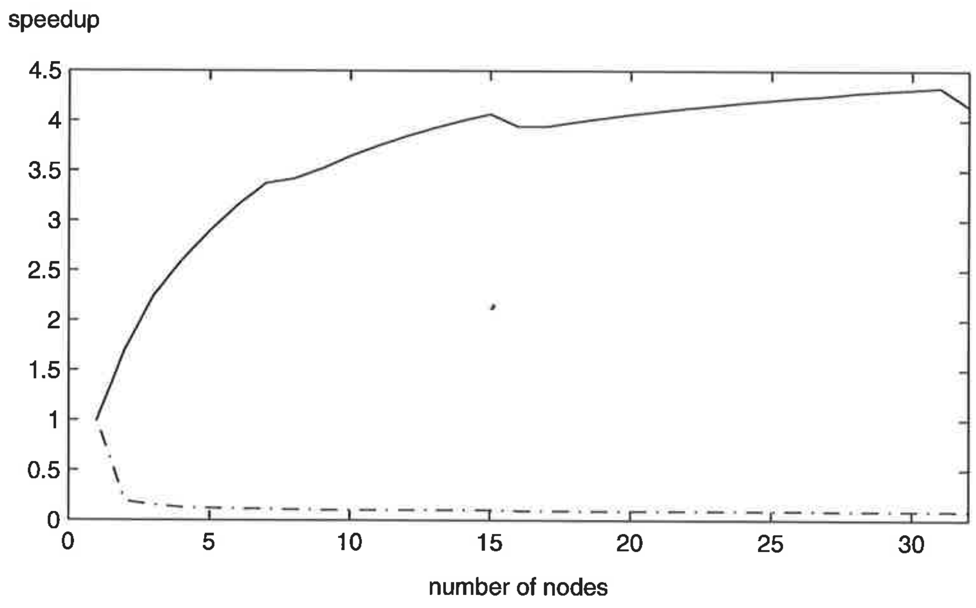


Figure 150. Speedup for optimised code for `sum.Adl` on 1 to 32 crossbar-connected nodes under the high-cost (solid-line) and low-cost (dashed-line) model of memory management with an input vector of 4096 words.

significant leaving the approximation:

$$\frac{T_p}{T_1} \approx \frac{n\beta}{t_{instr}c_{plus}n} = \frac{\beta}{t_{instr}c_{plus}}$$

Substituting $t_{instr}c_{plus} = 3ns$ and $\beta = 24ns$ we get $24ns/3ns$ which gives an asymptotic speedup of 0.125, clearly the small size of the binary operation + precludes any speedup for optimised `sum.Adl` on our model. Next we examine the performance of reduce with a larger binary operator.

reduce with maximum-segment-sum The key factor contributing to poor performance in the previous example is the small cost of the operation + compared to the cost of distributing its data. The imbalance in the costs of computation and communication can be improved, by improving the computation to communication ratio. Such a situation arises when a slightly more computationally expensive binary operation is used, such $\oplus$ in the Maximum-segment-sum algorithm in figure 151. Maximum-segment-sum calculates the maximum sum of any contiguous sub-sequence of a list of numbers. The possible presence of negative numbers makes this operation non-trivial. Computationally, the structure of maximum-segment-sum is a reduce with a binary operator $\oplus$ over tuples, followed by a projection on the final tuple.

The source code for maximum-segment-sum with reduce is shown in figure 151, while figure 152 shows parallelised optimised BMF code for `mss_reduce.Adl`.

The performance of this program, when run on varying numbers of processors, with varying quantities of input data, under the low-cost model, on a crossbar-connected machine, is shown in figure 153.

No point in the graph shows more than moderate speedup and this speedup does not appear to be growing strongly with data-size. This observation is in line the statically derived limit for this program running on a crossbar-connected machine, given by substituting $c_{\oplus} = 190$ into equation 68:

$$\frac{T_p}{T_1} \approx \frac{1}{p} + \frac{(2\log p)t_{link\_latency} + ((1-\frac{1}{p})n\beta + (1-\frac{1}{p})\beta}{190t_{instr}(n-1)}$$

With large p and n the above can be approximated:

$$\frac{T_p}{T_1} \approx \frac{n\beta}{190t_{instr}n} = \frac{\beta}{190t_{instr}}$$

which for our architectural model is equal to $24ns/190ns$, giving a maximum speedup of 7.9, in line with the performance measures shown in figure 153. Note that a crossbar is an ideal interconnect for a distributed memory parallel platform. If we move

```
main a: vof int :=
  let
    plus (x,y) := x + y;
    max (x,y) := if (x > y) then x else y endif;
    oplus ((mssx,misx,mcsx,tsx),(mssy,misy,mcsy,tsy)) :=
       (max(max(mssx,mssy),plus(mcsx,misy)),
        max(misx,plus(tsx,misy)),
        max(plus(mcsx,tsy),mcsy),tsx+tsy);
    f x :=
        let
          mx0 := max(x,0)
        in
          (mx0,mx0,mx0,x)
        endlet;
    first (mss,mis,mcs,ts) := mss
  in
    first(reduce(oplus,(0,0,0,0),map(f,a)))
  endlet
```

Figure 151. Source code for `mss_reduce.Adl` a program that calculates partial-maximum-segment-sums over an input vector. The binary operator used is `oplus`

$$
\begin{array}{l}
{}^4\pi_1 \cdot \\
\quad \oplus/^{\|}_{(0,0,0,0)^\circ} \cdot \\
\quad \oplus/_{(0,0,0,0)^\circ}) *^{\|} \cdot \\
\quad ((\pi_1, \pi_1, \pi_1, \pi_2)^\circ \cdot (\mathsf{if}(> \cdot (\mathsf{id}, 0)^\circ, \mathsf{id}, 0), \mathsf{id})^\circ) * *^{\|} \cdot \mathsf{split}_p \\
\textbf{\textit{where}} \\
\quad \oplus = \begin{array}[t]{l}
\mathsf{if}(> \cdot (\mathsf{if}(> \cdot ({}^8\pi_1, {}^8\pi_2)^\circ, {}^8\pi_1, {}^8\pi_2), + \cdot ({}^8\pi_3, {}^8\pi_4)^\circ)^\circ, \\
\qquad \mathsf{if}(> \cdot ({}^8\pi_1, {}^8\pi_2)^\circ, {}^8\pi_1, {}^8\pi_2), + \cdot ({}^8\pi_3, {}^8\pi_4)^\circ), \\
\mathsf{if}(> \cdot ({}^8\pi_5, + \cdot ({}^8\pi_6, {}^8\pi_4)^\circ)^\circ, {}^8\pi_5, + \cdot ({}^8\pi_6, {}^8\pi_4)^\circ), \\
\mathsf{if}(> \cdot (+ \cdot ({}^8\pi_3, {}^8\pi_7)^\circ, {}^8\pi_8)^\circ, \\
+ \cdot ({}^8\pi_3, {}^8\pi_7)^\circ, {}^8\pi_8), \\
+ \cdot ({}^8\pi_6, {}^8\pi_7)^\circ)^\circ \cdot \\
({}^4\pi_1 \cdot \pi_1, {}^4\pi_1 \cdot \pi_2, {}^4\pi_3 \cdot \pi_1, {}^4\pi_2 \cdot \pi_2, {}^4\pi_2 \cdot \pi_1, {}^4\pi_4 \cdot \pi_1, {}^4\pi_4 \cdot \pi_2, {}^4\pi_3 \cdot \pi_2)^\circ
\end{array}
\end{array}
$$

Figure 152. Parallel optimised version of `mss_reduce.Adl`

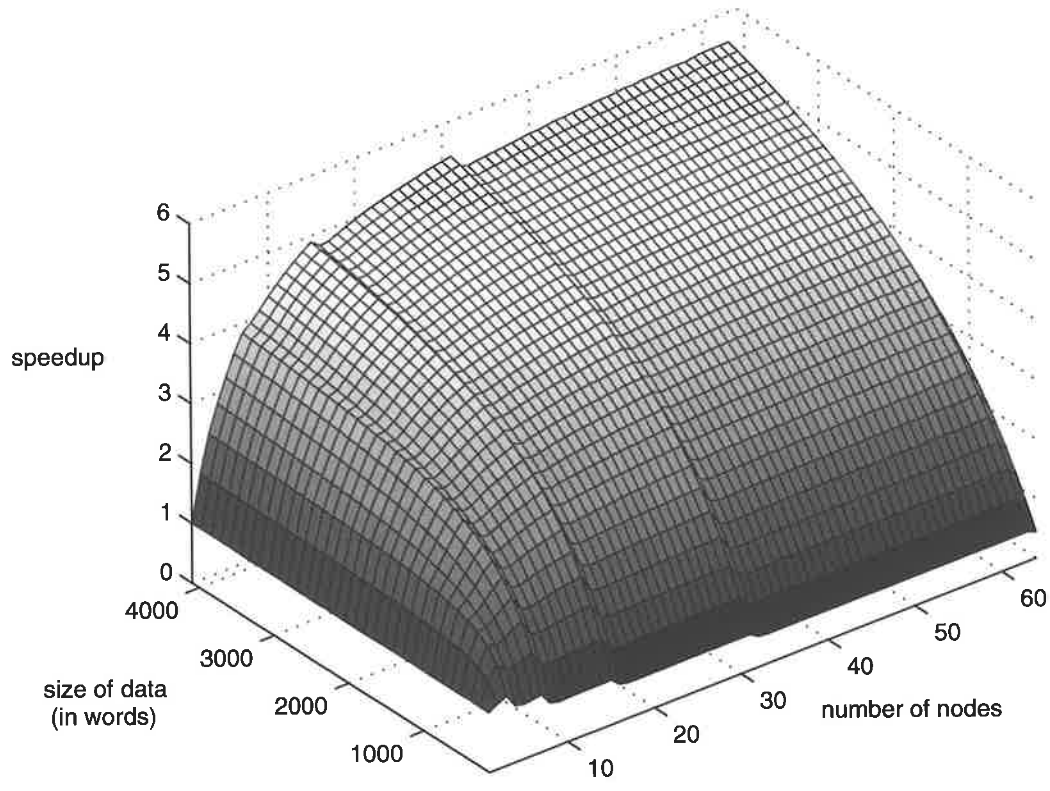


Figure 153. Speedup for optimised `mss_reduce.Adl` with various numbers of nodes and data sizes on a 64 node cross-bar connected machine under the low-cost memory management model.

to a more scalable interconnect, such as a mesh, we attain the performance shown in figure 154. This performance, affected by longer latencies in a mesh interconnect.

It should be stressed that the above results do not preclude applications that use reduce from exhibiting good speedup. Programs that:

1. have a long stage of parallel computation prior to a reduction or,

2. generate, as part of the parallel computation, most the data to be reduced on each node.

can usefully exploit large amounts of parallelism. Experiments 4 (`finite_diff_iter.Adl`) and 5 (`remote.Adl`), described shortly, fit these respective categories.

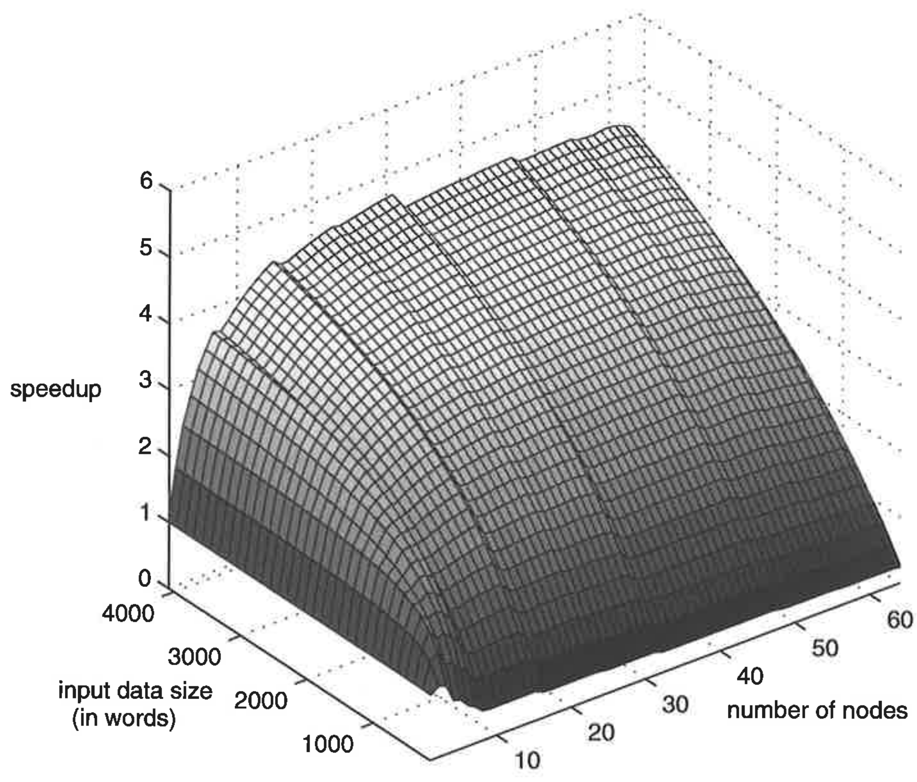


Figure 154. Speedup for optimised `mss_reduce.Adl` with various numbers of nodes and data sizes on a 64 node mesh-connected machine.

6.4.5.1 Experiment 3: Scan with maximum-segment-sum

For any given operator, scan is more communications-intensive than reduce. Here, we use the best parallelisation scheme for scan (described on page 205) to parallelise both translator and optimiser code for `mss.Adl` (shown in figure 98 on page 159). The parallelised version of the optimiser code for `mss.Adl` is shown in figure 155 (the parallelised translator code is substantially longer).

Figure 156 shows the execution times for translator and optimiser code under both the high-cost (solid line) and low-cost (dashed line) models of memory management on between 1 and 32 crossbar-connected nodes. As with previous experiments, the optimiser code is substantially faster than the translator code running on the corresponding memory management model. However, the cost of memory management weighs more heavily in this example than in previous ones. In fact, the cost of the optimised code for `mss.Adl` on a single node, under the high-cost memory model, exceeds that of the corresponding unoptimised code under the low-cost memory model. This is probably due, in large part, to the amount of local

$$
\begin{array}{l}
+\!\!+ /^{\parallel} \cdot ((^{4}\pi_1)*) *^{\parallel} \cdot g \cdot \\
(\oplus /\!\!/^{\parallel} \cdot \mathit{init} \cdot (\mathit{last}) *^{\parallel}, \mathsf{id})^{\circ} \cdot (\oplus /\!\!/) *^{\parallel} \cdot \\
((({}^{2}\pi_1, {}^{2}\pi_1, {}^{2}\pi_1, {}^{2}\pi_2)^{\circ} \cdot (\mathsf{if}(> \cdot (\mathsf{id}, 0)^{\circ}, \mathsf{id}, 0), \mathsf{id})^{\circ})*) *^{\parallel} \cdot \mathsf{split}_{32} \\
\textbf{\textit{where}} \\
\quad \begin{array}{lcl}
\oplus & = & \textit{from previous example} \\
g & = & +\!\!+^{\parallel} \cdot (\mathsf{split}_1 \cdot ! \cdot ({}^{2}\pi_2, 0)^{\circ}, (\odot) *^{\parallel} \cdot \mathsf{zip}\| \cdot ({}^{2}\pi_1, \mathit{tail} \cdot {}^{2}\pi_2)^{\circ})^{\circ} \\
\odot & = & (\oplus) * \cdot \mathsf{distl} \cdot ({}^{2}\pi_1, {}^{2}\pi_2)^{\circ} \\
\mathit{tail} & = & \mathsf{select}^{\parallel} \cdot (\mathsf{id}, (+ \cdot (1, \mathsf{id})^{\circ}) * \cdot \mathsf{iota} \cdot - \cdot (\# \cdot \mathsf{id}, 1)^{\circ})^{\circ} \\
\mathit{init} & = & \mathsf{select}^{\parallel} \cdot (\mathsf{id}, \mathsf{iota} \cdot - \cdot (\#, 1)^{\circ})^{\circ} \\
\mathit{last} & = & ! \cdot (\mathsf{id}, - \cdot (\#, 1)^{\circ})^{\circ}
\end{array}
\end{array}
$$

Figure 155. parallelised optimiser code for `mss.Adl`

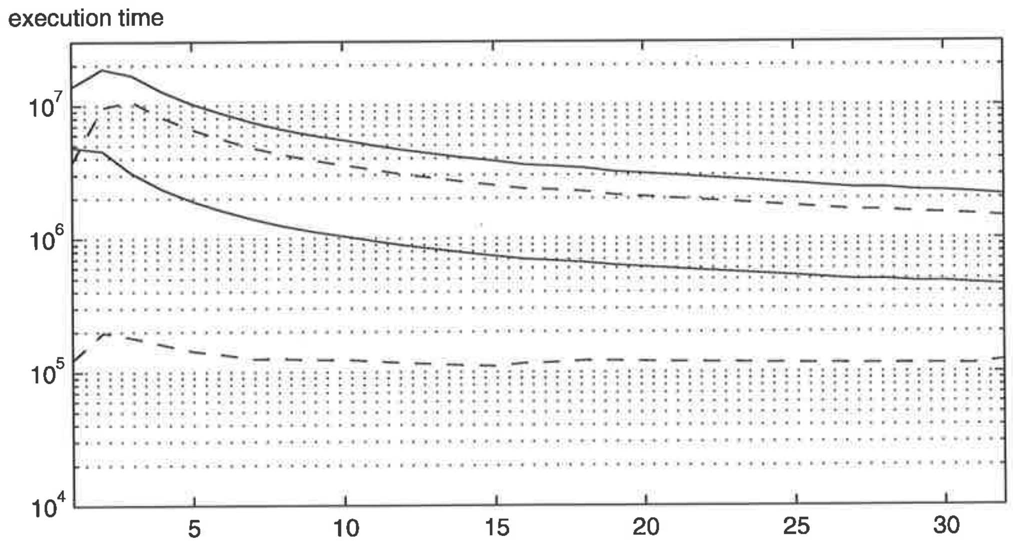


Figure 156. Execution times for translator and optimiser code for `mss.Adl` under both the high-cost (solid line) and (low-cost) models of memory management. The execution times for translator code are the two upper lines and for optimiser code the two lower lines.

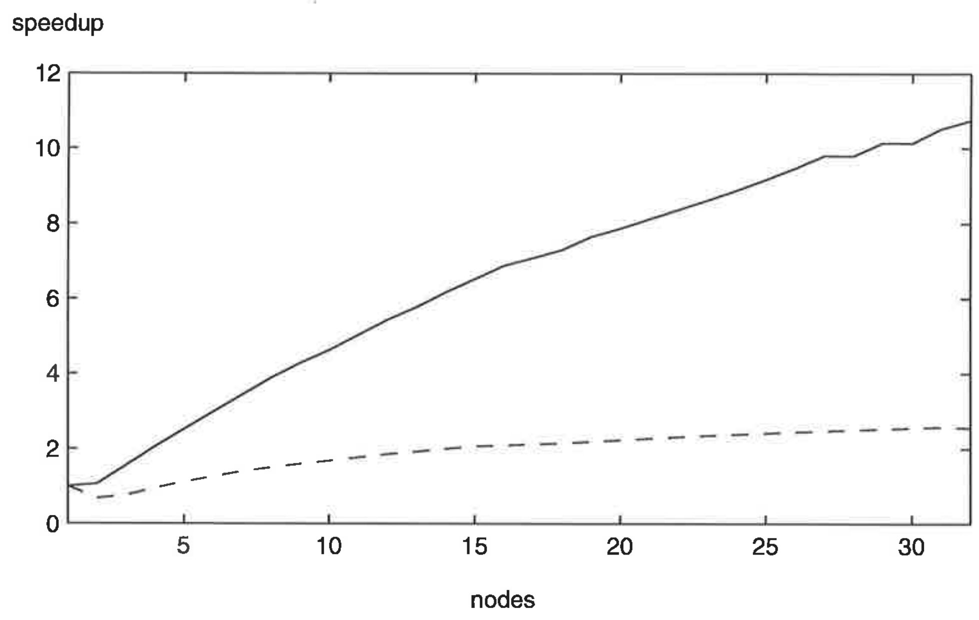


Figure 157. Speedups for optimised code for `mss.Adl`, running under the high (upper curve) and low-cost (lower curve), memory model when applied to a 4096 element input vector.

copying generated by the detailed code in the parallelised version[36] of `mss.Adl` in both optimised and non-optimised code. The high-cost memory management model magnifies this copying cost. Note that the relative cost of the optimised code drops as more nodes are used and the low communication costs of the optimised version become an important factor.

Reasonable speedups are attained under all regimes, except the optimised code under the low-cost model (bottom line), which shows negligible speedup.

Figure 157 shows the speedups for optimised code, running under the high and low-cost, memory model when applied to a 4096 element input vector. Speedup under the high-cost memory management model is moderately good. Speedup under the low-cost model is still quite poor. The speedup evident in the higher curve is attributable to the extra computation costs imposed by the high-cost memory management regime, which will be significantly reduced when memory management costs are optimised.

Figure 158 takes the low-cost curve from figure 157 and compares it with corresponding costs for input data vectors between 256 and 8192 elements long. As with `mss_reduce.Adl` the speedup does grow with data-size but not at a particularly

[36]In this case the parallelised version running on a single node.

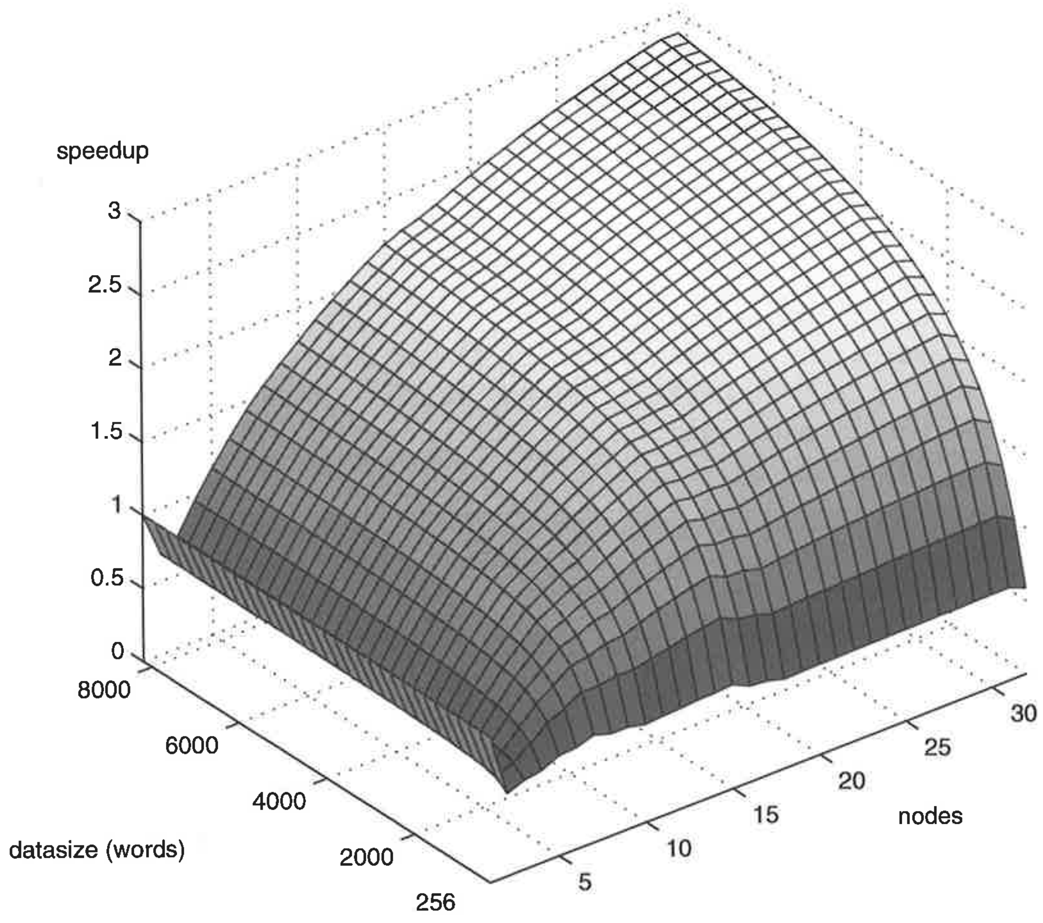


Figure 158. Speedup of the optimiser code for `mss.Adl` when run on between 1 and 32 crossbar-connected nodes and between 256 and 8192 element input vectors.

fast rate. Worthwhile speedups are not going to be attained under the low-cost model by simply increasing the data-size.

The modest speedup is, again due to the relatively high cost of inter-node bandwidth, caused by the low ratio of computation to communication in this problem. If we compare figure 153 to figure 158 we see the performance of the second is slightly worse. This is due to scan, even when implemented using a good strategy, generating more communication than reduce. The modest difference between these graphs is testimony to effectiveness of the strategy for scan[37]

The fundamental problem with `mss.Adl` and the other programs we have described in our experiments, thus far, is that the ratio of computation to communication is

[37]This indirectly highlights a weakness in the use of virtual processors. With virtual processors, partitions are all of size one, so the strategy of transmitting only the upper value of each partition to the neighbouring partition is of no benefit.

```
main a: vof int :=
  let
    addone x := x + 1;
    first (x,y) := x;
    ia := iota (#a);
    f (xs,y) :=
      let
        lxs := #xs;
        stencil x := xs!x + xs!((x-1) mod lxs)  +
                     xs!((x+1)  mod lxs);
        nxs := map(stencil,ia);
        ny := addone y
      in
        (nxs,ny)
      endlet;
    p(xs,y) := y < 20
  in
    first(while(f,p,(a ,0)))
  endlet
```

Figure 159. `finite_diff.Adl` an Adl program to perform a simple finite-difference computation.

a constant, irrespective of grain-size. Computations in this category are bandwidth-limited in their parallel performance.

6.4.6 Experiment 4: `finite_diff_iter.Adl`

In this experiment, and the next, we test programs where the ratio of communications to computation improves with increasing problem size.

One way to achieve a healthy computation to communications ratio is to iterate over pre-distributed data, performing computations on, mostly, local data during each iteration. Figure 159 shows the source-code of `finite_diff_iter.Adl`, a program that fits this desirable pattern of computation[38]. The loop body is a simple one-dimensional stencil operation, with each element performing a simple calculation based on its own value and its left and right neighbours.

[38]This program is a modified version of `finite_diff.Adl` shown in figure 100 in chapter 5. The important difference is the stencil operation is embedded in a while loop. The body of the while loop avoids the use of iota to allocate a new vector `finite_diff.Adl` as this would be a moderately expensive operation in a distributed context.

$$
\begin{array}{l}
\pi_1 \cdot \pi_2 \\
(\ \mathsf{id}, \\
\quad \pi_2 \cdot \mathsf{while}(\ (\ \pi_1, \\
\qquad\qquad\qquad (\pi_2 \cdot \pi_1, \pi_2)^\circ \cdot \mathsf{id} \cdot (\mathsf{id}, + \cdot (\pi_2, 1)^\circ \cdot (\mathsf{id}, \pi_2 \cdot \pi_2 \cdot \pi_1 \cdot \pi_1)^\circ)^\circ \cdot \\
\qquad\qquad\qquad (\mathsf{id},\ +\!\!+ /^{\|} \cdot \\
\qquad\qquad\qquad\qquad ((+ \cdot (\ + \cdot (\ ! \cdot (\pi_1 \cdot \pi_2 \cdot \pi_1 \cdot \pi_1, \pi_2)^\circ, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad ! \cdot (\pi_1 \cdot \pi_2 \cdot \pi_1 \cdot \pi_1, \mathsf{mod} \cdot (- \cdot (\pi_2, 1)^\circ, \pi_2 \cdot \pi_1)^\circ)^\circ)^\circ, \\
\qquad\qquad\qquad\qquad\qquad\quad ! \cdot (\pi_1 \cdot \pi_2 \cdot \pi_1 \cdot \pi_1, \mathsf{mod} \cdot (+ \cdot (\pi_2, 1)^\circ, \pi_2 \cdot \pi_1)^\circ)^\circ)^\circ)*) *^{\|} \cdot \\
\qquad\qquad\qquad (\mathsf{distl}) *^{\|} \cdot \mathsf{distl}^{\|} \cdot (\mathsf{id}, \pi_2 \cdot \pi_1 \cdot \pi_1)^\circ)^\circ \cdot (\mathsf{id}, \# \cdot \pi_1 \cdot \pi_2)^\circ)^\circ \\
\qquad\qquad < \cdot(\pi_2 \cdot \pi_2, 20)^\circ) \cdot \\
\quad (\mathsf{id}, (\pi_1, 0)^\circ)^\circ)^\circ \cdot \mathsf{id} \cdot (\mathsf{id}, \mathsf{split}_p \cdot \mathsf{iota} \cdot \# \cdot \mathsf{id})^\circ \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (a)
\end{array}
$$

$$
\begin{array}{l}
+\!\!+ /^{\|} \cdot \pi_1 \\
\mathsf{while}(\ (\ {}^3\pi_1, \\
\qquad\quad (((+ \cdot (+ \cdot \pi_1, \pi_2)^\circ)*) *^{\|} \cdot \\
\qquad\qquad\quad (\mathsf{zip}) *^{\|} \cdot \mathsf{zip}^{\|} \cdot ((\mathsf{zip}) *^{\|} \cdot \mathsf{zip}^{\|} \cdot ({}^3\pi_1, \mathit{rshift})^\circ, \mathit{lshift})^\circ \cdot \\
\qquad\qquad\quad ({}^3\pi_2, \pi_2 \cdot^3 \pi_1, \# \cdot^3 \pi_2)^\circ, \\
\qquad\qquad\quad + \cdot (\pi_1 \cdot^3 \pi_3, 1)^\circ)^\circ)^\circ \cdot \\
\qquad\quad (\pi_1, \pi_1 \cdot \pi_2, (\pi_2, \mathsf{id})^\circ \cdot \pi_2)^\circ, \\
\qquad < \cdot(\pi_2, 20)^\circ \cdot \pi_2) \\
(\mathsf{id}, (\pi_1, 0)^\circ)^\circ \cdot (\mathsf{split}_p \cdot \mathsf{id}, \mathsf{split}_p \cdot \mathsf{iota} \cdot \#)^\circ \\
\qquad\qquad\qquad\qquad\qquad\qquad (b)
\end{array}
$$

Figure 160. Parallelised translator code (part (a)) and optimiser code (part (b)) for `finite_diff_iter.Adl`.

Figure 160 shows parallelised translator code (part (a)) and optimiser code (part (b)) for `finite_diff_iter.Adl`. The translator code relies on indexing functions (!) to extract the desired vector elements for each stencil operation. A parallelised distl operation is used to transport copies of vectors required by these operations. The main index vector, generated by the iota function, remains distributed throughout the loop. Aside from the myriad of addressing functions the translator code is quite straightforward but the distl operations exact a cost in terms of efficiency.

The optimiser code, in part (b) of the figure, splits both the index vector and the input vector. The select functions that resided in the loop body have been replaced by code for *lshift* and *rshift*. These functions contain detailed hand-generated BMF code implementing a parallel circular-shift[39]. In future, these functions are likely be

[39]At the time of writing, distrselect is not implemented on the simulator.

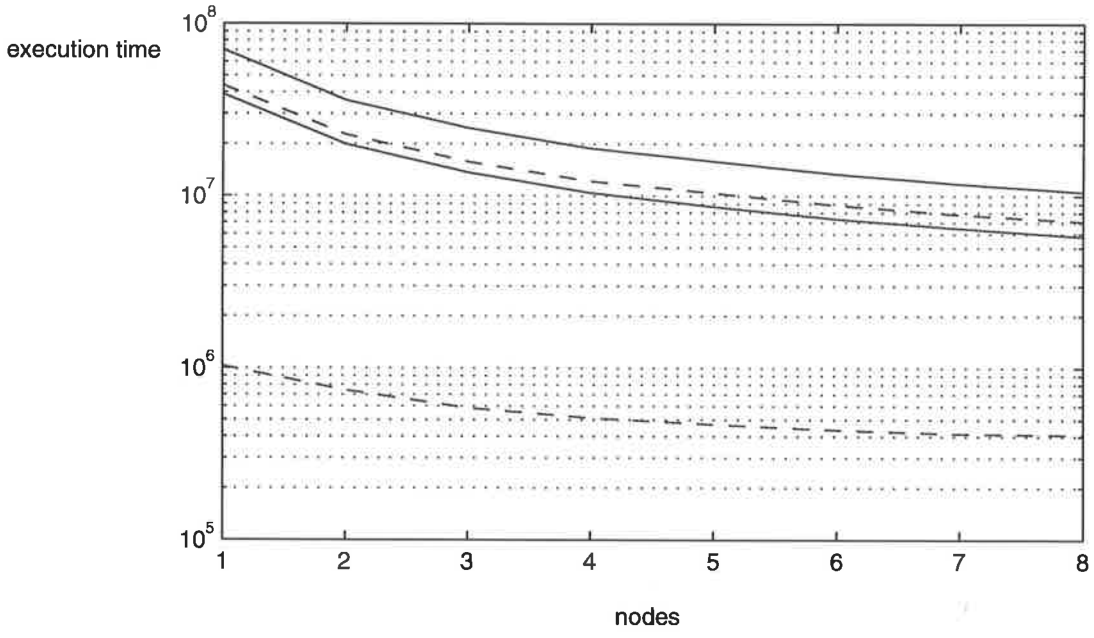


Figure 161. Parallel translator and optimiser code for 20 iterations of `finite_diff_iter.Adl` under both the fast and slow memory-management regimes on between 1 and 8 crossbar connected nodes with 256 input values

replaced by specialised primitives.

A-priori we would expect the optimiser code to run faster because it performs less communication inside the loop. This is borne out by figure 161. Due to the time taken by the simulator to run translator code, the number of processors was limited to eight and the size of input data was limited to 256. The number of iterations of the while loop was fixed at 20. The top curve, is the performance of translator code under the high-cost memory management regime. As with previous examples, speedup is good, again due to a high computation to communication ratio. The second-from-top curve shows the performance of translator code under the low-cost memory management regime. The third curve is the performance of optimiser code under the high-cost memory-management regime. This code shows a substantial, though not overwhelming improvement over the corresponding parallel translator code. It may be that the data-size is too small to open up a substantial performance gap. The bottom curve is the performance of the optimiser code under the low-cost memory management model. This code is, by far, the most efficient but it also exhibits the lowest speedup. As with previous examples the low speedup is attributable to the low computation to communication ratio in this program at this small data-size.

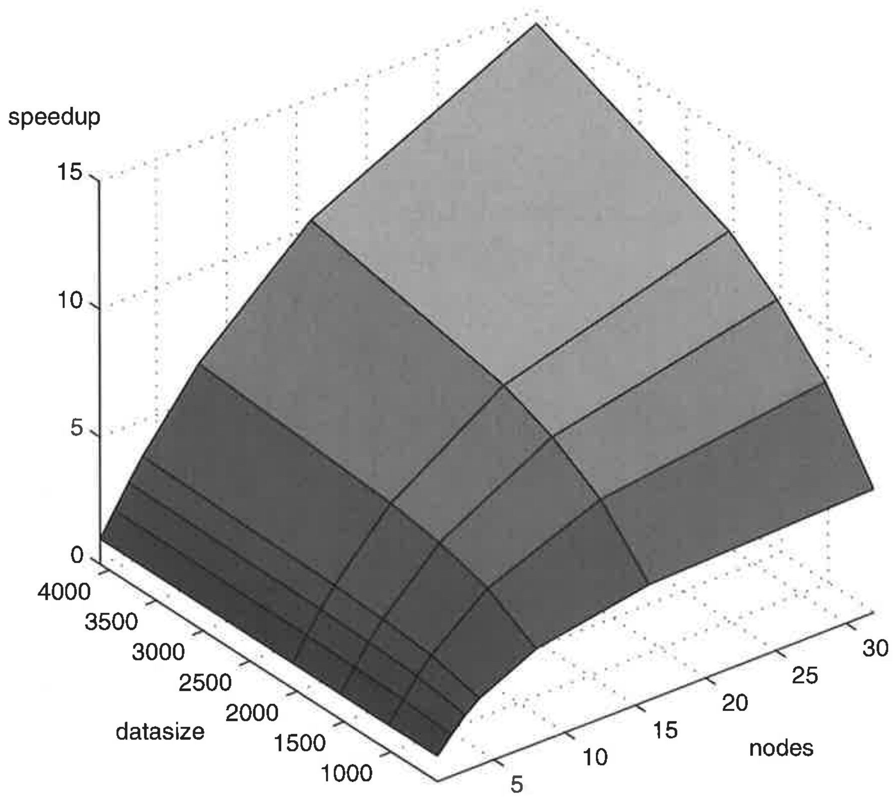
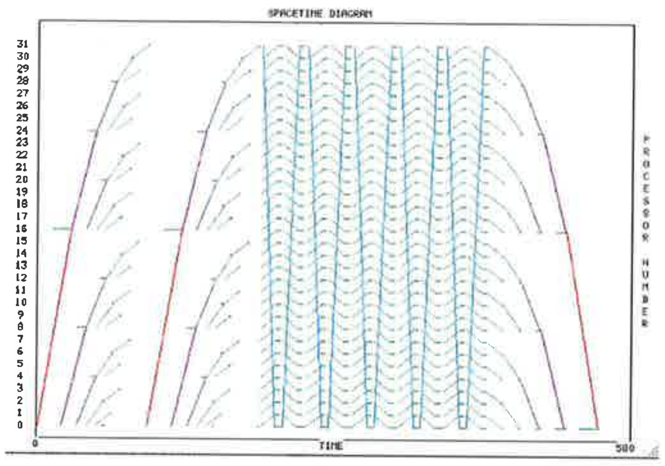


Figure 162. Speedup for `finite_diff_iter.Adl` executing with 20 loop iterations for 1, 2, 3, 4, 8, 16 and 32 crossbar-connected nodes on different data-sizes.

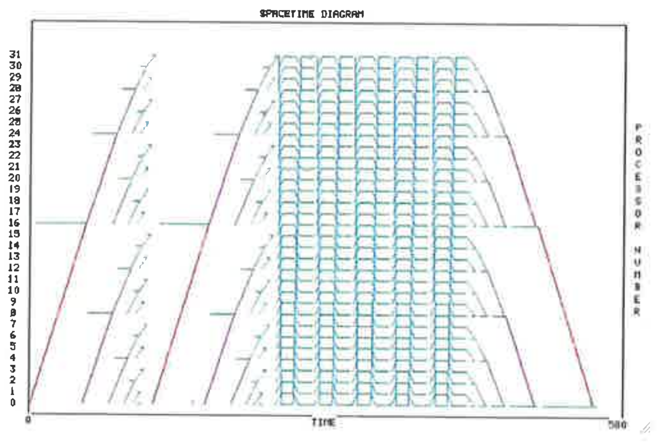
Fortunately, in the case of `finite_diff_iter.Adl`, this ratio improves substantially with data-size.

Figure 162 shows the speedup for `finite_diff_iter.Adl` running under the fast model of memory management for a range of data sizes. The cost of running the simulator made it impractical to collect as many data-points as in previous experiments, but a trend is still discernible. The curve is much steeper, and the absolute speedups are much greater, than for the previous examples in this chapter. Note that the curve slowly levels off both in the direction of increasing nodes and of increasing data-size. This levelling-off is explained by the fact, that even for very large data, for a fixed number of iterations, there is a limit to speedup attainable for `finite_diff.Adl`.

The existence of this limit could be demonstrated through static analysis, as with previous examples, but a strong intuitive understanding is gained by comparing traces for different input data sizes. Figure 163 contains two space-time traces for parallel optimised `finite_diff_iter.Adl`, running on 32 crossbar-connected nodes,



(*a*)



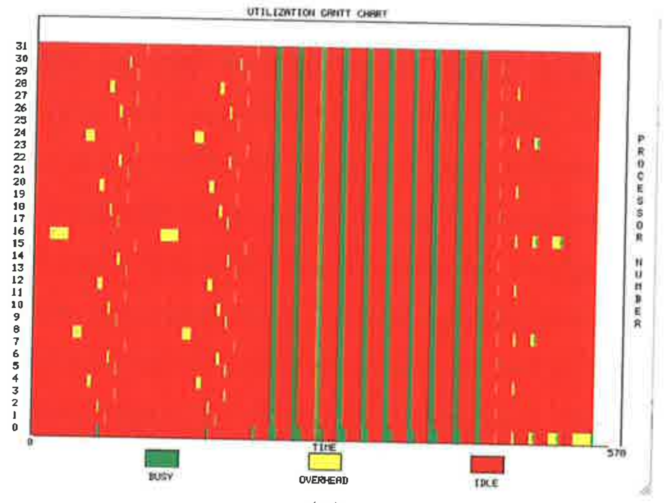
(*b*)

Figure 163. Space-time traces for parallel optimiser code for `finite_diff_iter.Adl` running on 32 nodes of a crossbar-connected machine under the low-cost memory management regime. Part (a) is the trace for a 512 element input value. Part (b) is the corresponding, scaled-down, trace for a 1024 element input value. Note that as the size of the input value increases the relative significance of the loop (the central part of the trace) decreases.

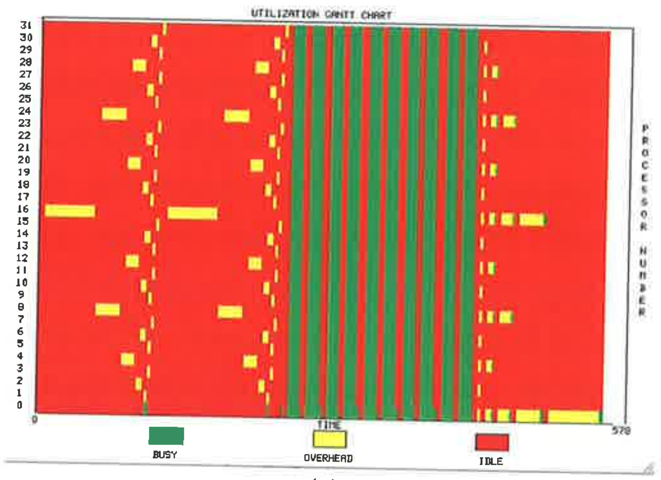
collected by the simulator. Time is on the horizontal scale and node numbers are on the vertical scale. Inter-node communication is represented by angled lines. In both diagrams there is substantial communication at the start and the end of the trace as the initial array is distributed[40] and re-coalesced. There are smaller amounts of communication in the loop as single values are sent between adjacent nodes in an end-around pattern by the *lshift* and *rshift* functions. The horizontal lines in the traces represent receive-overhead and computation. Part (a) is the trace generated for an input array of 512 elements. Part (b) is the trace generated for an input array of 1024 elements. The length of the second trace was scaled down by a factor of more than two for the sake of comparison. Ideally, with increasing data-size, the cost of computation, relative to the cost of communication, should increase. Unfortunately, this is only the case for the code executing *inside* the central loop. Outside of the loop the relative cost of communication is slightly more for part (b) than part (a). In fact, the relative significance of the loop in the program has decreased in line with the decreased relative cost of communications inside the loop. This change in the relative significance of communications inside and outside of the loop can be clearly seen in the processor-utilisation charts in figure 164. These charts are derived from the same traces as the charts in figure 163. The bright green areas represent compute-time[41]. The vertical green bars represent the useful computation that takes place between invocations of *lshift* and *rshift* inside the loop. These bars are much thicker in part (b) than part (a). With large data, computation-time dominates the loop. Outside the loop, the relative size of the red and yellow areas are not at all diminished by increasing data-size. They grow and shrink stubbornly in tandem with the input data. However, they are fixed with respect to the number of iterations in the loop. If the central loop had 1000 iterations, rather than five, then red areas at either end would be dwarfed by the mostly-green area in the middle. In summary, utilisation inside the loop increases with the size of input data, and the relative significance of the loop grows with the number of iterations. Taken together, these two observations mean that very good speedup can be attained for `finite_diff_iter.Adl` by a combination of large data and a large number of iterations.

[40]In fact, two arrays are distributed. One corresponding to each instance of split in figure 160. There is scope for piggybacking of these communications; something which future versions of the paralleliser can address.

[41]With perfect speedup the whole graph would be green.



(a)



(b)

Figure 164. Processor-utilisation (Gantt) charts for parallel optimiser code for `finite_diff_iter.Adl` running on 32 nodes of a crossbar-connected machine under the low-cost memory management regime. Part (a) is the trace for a 512 element input value. Part (b) is the corresponding, scaled-down, trace for a 1024 element input value. Red areas represent idle processor time, yellow areas represent receive-overhead, green areas represent actual compute-time.

```
main a: vof int :=
  let
    f x :=
      let
        add(x ,y) := x+y;
        abs x := if x<0 then -x else x endif;
        dist y := abs(x-y)
      in
        reduce(add, 0, map(dist, a))
      endlet
  in
    map(f, a)
  endlet
```

Figure 165. Source code for `remote.Adl`.

6.4.7 Experiment 5: `remote.Adl`

This, final, example describes an application performing a convolution where every value in an input vector must interact with every other input value. Intuitively, there are two things that we might expect to see from such an application

- Relatively good performance of translator code due to a match between the distribution performed in the application and the distribution performed by translator.
- Good speedup, given a large enough grain size, because one copy of an array sent to each node is enough to fuel a large amount of local distribution and computation.

Figure 165 shows the source code for `remote.Adl`, a program that calculates the sum of the absolute difference between each value and every other value in an input vector. As an example, if `remote.Adl` is given the input vector $[1, 2, 3, 4, 5]$ its output will be $[10, 7, 6, 7, 10]$.

Figure 166 shows the parallelised translator and optimiser code for `remote.Adl`. The translator code (in part (a)), as with all translator code, distributes all values in scope to each function. This leads to unnecessarily detailed code and inefficient execution. The optimiser code (in part (b)), still performs some distribution, albeit of smaller values, using either a combination of repeat and zip or distl. This distribution is required by the problem being solved, a convolution.

$$
\begin{array}{l}
+\!\!+ /^{\|} \cdot (\mathit{reduce\_exp} \cdot (\mathsf{id}, (\mathit{abs} \cdot (\mathsf{id}, \mathit{minus})^{\circ}) * \cdot \mathsf{distl} \cdot (\mathsf{id}, \pi_1)^{\circ})^{\circ} \cdot \mathsf{id}) * *^{\|} \cdot \\
(\mathsf{distl}) *^{\|} \cdot \mathsf{distl}^{\|} \cdot (\mathsf{id}, \mathsf{split}_p \cdot \mathsf{id})^{\circ} \\
\textbf{\textit{where}} \\
\quad \begin{array}{lcl}
\mathit{reduce\_exp} & = & \mathsf{if}(\neq \cdot (0, \# \cdot \pi_2)^{\circ}, \oplus/ \cdot \mathsf{distl}, 0 \cdot \pi_1) \\
\oplus & = & + \cdot (\pi_1 \cdot \pi_2, \pi_2 \cdot \pi_2)^{\circ} \cdot (\pi_1 \cdot \pi_1, (\pi_2 \cdot \pi_1, \pi_2 \cdot \pi_2)^{\circ})^{\circ})^{\circ} \\
\mathit{abs} & = & \mathsf{if}(< \cdot (\pi_2, 0)^{\circ}, u- \cdot \pi_2, \pi_2) \\
u- & = & \textit{primitive operation (unary minus)} \\
\mathit{minus} & = & - \cdot (\pi_2 \cdot \pi_1, \pi_2)^{\circ}
\end{array}
\end{array}
$$

(a)

$$
\begin{array}{l}
+\!\!+ /^{\|} \cdot \\
(+/_0 \cdot (\mathsf{if}(< \cdot (-, 0)^{\circ}, u- \cdot -, -)) * \cdot \mathsf{distl}) * *^{\|} \cdot \\
(\mathsf{zip}) *^{\|} \cdot ((\pi_1, \mathsf{repeat} \cdot (\pi_2, \# \cdot \pi_1)^{\circ})^{\circ}) *^{\|} \cdot \\
\mathsf{zip}^{\|} \cdot (\pi_1, \mathsf{repeat}^{\|} \cdot (\pi_2, \# \cdot \pi_1)^{\circ})^{\circ} \cdot (\mathsf{split}_p, \mathsf{id})^{\circ}
\end{array}
$$

(b)

Figure 166. Parallelised translator code (part(a)) and optimiser code (part (b)) for `remote.Adl`.

Figure 167 shows the relative performance of translator and optimiser code for `remote.Adl`. Both translator and optimiser code was applied to an input vector of 320 elements under both the fast and slow memory management models on between one and eight crossbar-connected nodes. In contrast with previous examples, reasonable speedup is attained under all four combinations of conditions. This may be due to the relatively large input vector[42] and/or the parallelism inherent in the application. The top two curves represent the performance of translator code under the slow memory management regime (solid line) and the fast memory management regime (dashed-line). The bottom two lines represent the performance of optimiser code. The gap between the optimiser and translator code under the slow memory management regime is significant but not extreme. This is consistent with our intuition that a certain amount of distribution is inherent in this application and not liable to be optimised away.

It remains to be determined if the good speedups exhibited by optimiser code persist when more nodes are utilised. Figure 168 shows the speedup gained by running the optimised version of `remote.Adl` on between one and 32 nodes of a crossbar-connected machine under both the fast and the slow memory management

[42]320 elements is large for this problem because of the way the vector is copied in `remote.Adl`.

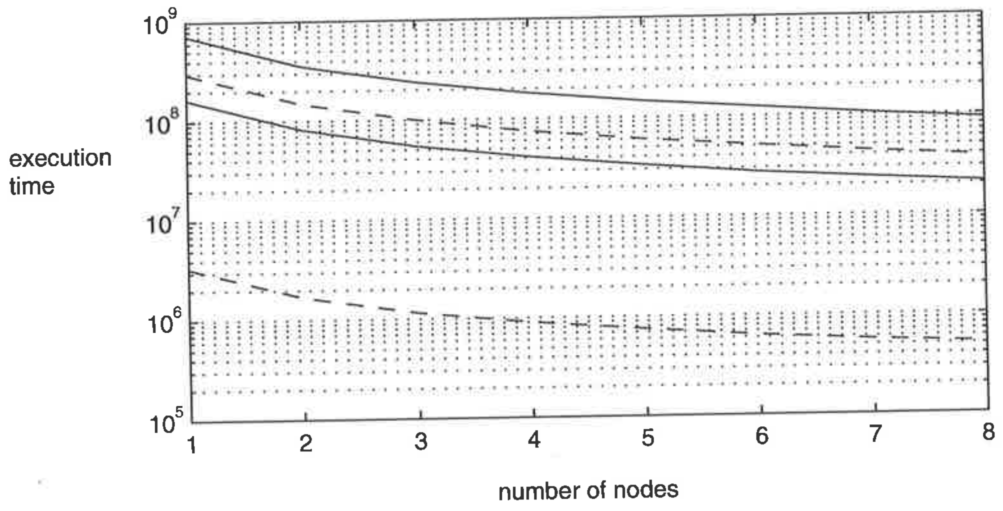


Figure 167. The performance of `remote.Adl` running on between one and eight crossbar-connected nodes with 320 input values. The top two lines represent the performance of the translator code under the high cost (solid line) and low cost (dashed line) memory management models. The bottom two lines shows the corresponding performance of the optimiser code under the high and low cost memory management models.

model with an input vector of 320 elements. Under the high-cost model speedup is very close to linear for the whole range of processors. The low cost model has near linear speedup for small numbers of processors but levels off for larger numbers. This levelling-off appears characteristic of the impact of a slightly low and decreasing computation to communication ratio. For `remote.Adl` this ratio can be improved by increasing the data size. We would expect to see better speedup at higher numbers of nodes for input with larger data.

To test this relationship we ran a series of experiments with various data sizes to plot the surface seen in figure 169. For 32 processors, speedup grows quite rapidly with data-size. For smaller numbers of processors this growth tails off as the speedup gets closer to the maximum potential for that number of processors[43]. This data indicates that good speedup is achievable, for a given number of nodes, if enough input data used.

This completes the presentation of our experimental results. The next section reviews related work and, after that, the findings of this chapter and future work are

[43]Because the model does not take into account on-node cache performance, and virtual memory performance there is no scope for achieving speedup beyond linear speedup as there is on some real machines.

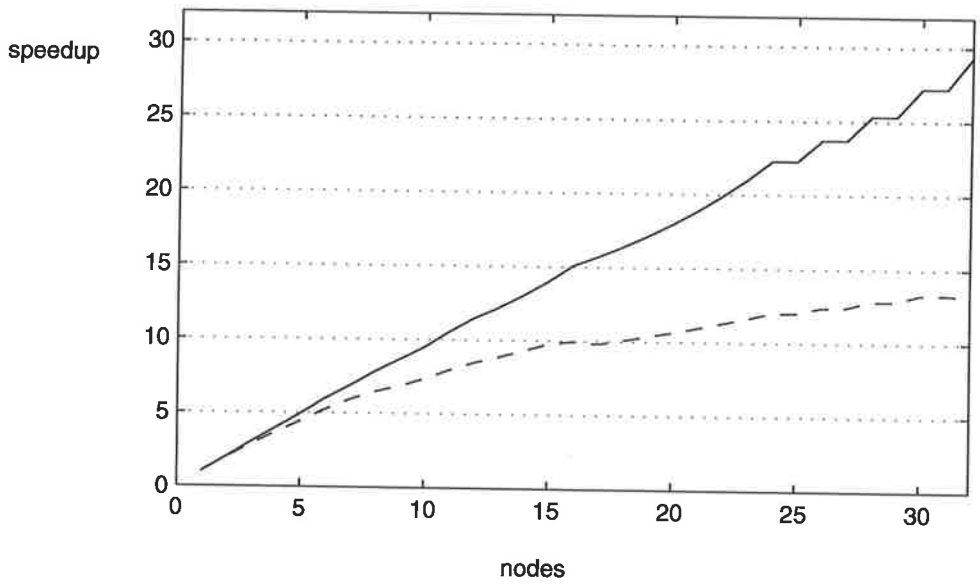


Figure 168. Speedup for `remote.Adl` running on between 1 and 32 nodes of a crossbar-connected machine. The upper curve shows the speedup under the high-cost memory management regime. The lower curve shows the speedup under the low cost regime. The input vector was 320 elements long.

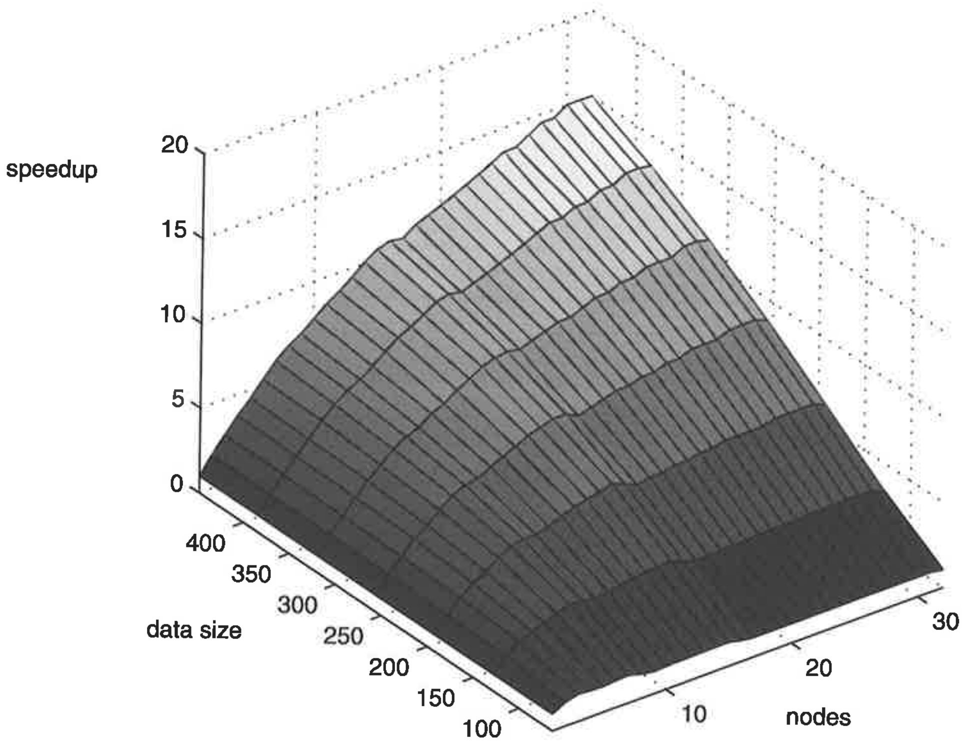


Figure 169. Speedup for `remote.Adl` running on between 1 and 32 nodes with input data between 64 elements and 448 elements.

presented.

6.5 Related work

Related work can be, very roughly, divided into work related by the use of transformations and work related by the use of parallel skeletons. There is substantial overlap, so some work mentioned below straddles both categories.

Related transformational work The use of equational transformation as a means of developing efficient parallel functional programs is not a new concept. Backus[13] cited amenability to parallel execution and algebraic transformation as two strengths of functional programming. Proponents of functional programming models such as Algorithmic Skeletons[42], Divacon[28], MOA[70] and Scan-Vector[92] have emphasised the desirability of equational transformation as a tool for programming and program optimisation.

The use of equational transformation for parallel programming in BMF is relatively common. A number of authors[126, 58, 35, 103] have highlighted the usefulness and generality of BMF as a medium for parallel programming. Gorlatch et. al.[64, 63] and Hu[78](summary) have explored mechanisable formal methods for improving the performance of BMF programs with implicit parallel semantics.

A smaller amount of literature is devoted to the problem of algebraically propagating explicit parallel functions into an extant BMF program. Skillicorn[127, 125] outlined a technique where parallel operations can be progressively added to a BMF program with no net change to its semantics. The parallel operations have two interpretations:

- A functional interpretation (input to output mapping) which allows the operations to appear in the mathematical equalities used to algebraically transform programs.
- A separate parallel interpretation which describes the affect of the operation on a distributed architecture.

A similar method of derivation was also developed by Roe[120] to help target programs to both SIMD and MIMD architecture types. Partitionings can be generalised to arbitrary complexity and arbitrary datatypes. Pepper et. al[109] describe general schema's for partitioning. Jay[86] allows quite general decompositions of matrices in his proposal for the GoldFISh parallel programming language. The parallelisation process defined in this work is strongly influenced by the above work of Skillicorn and Roe.

In other, related, work Skillicorn and Cai[128] refined the derivation process by adding cost annotations to identities involving parallel operations. The annotations form part of a static cost model for making estimations of the efficiency of code while the parallel program is derived. Currently, the parallelisation process for the Adl project does not make use of a cost model, though the simulator used in our experiments provides a detailed dynamic model that a developer can use to tune the parallelised program.

Related skeletons work The idea of using algorithmic skeletons for the expression of parallelism was proposed by Murray Cole[34]. The Adl implementation exploits skeletal parallelism. The following contrasts the Adl implementation with three extant Skeleton implementations, SCL, the Anacleto compiler (P3L), and the more recent FAN Skeleton framework.

SCL (Structured Coordination Language)[43] focuses on the use of functional skeletons to coordinate communications between program components written in other language. Much emphasis is put on ensuring that data is aligned and transported to its required location. Parallelism and communications are more explicit in SCL than Adl meaning that more has to be specified by the programmer. On the other hand, there is more flexibility in the partitioning and alignment of data in SCL than Adl.

Anacleto[31], the prototype implementation of P3L[41] supports task and control as well as stream and data-parallel skeletons. P3L has a more explicit constrained indexing scheme for access to vectors inside the `map` skeleton than Adl[44]. Anacleto uses compile-time performance-analysis of code to drive aspects of the partitioning process[45]. P3L supports static arrays of fixed dimensionality and size. Adl supports

[44]In the main, it is the responsibility of the programmer to give the compiler a description of the values required within each invocation of a data-parallel construct, through parameter lists and annotations specifying the scattering, broadcast and multi-cast of array values. In Adl, the programmer is able to reference any value in scope at will. It is the compiler's task to rationalise the transport of these values.

[45]This amounts to a well-defined and constrained, iterative transformation process. More detailed transformation strategies were also explored using P3L as a medium[4].

dynamic nested one-dimensional arrays (vectors). Another significant difference is that P3L is built around an imperative core of C-code. Programmers are able to recycle variables so storage management issues that arise with the Adl project do not arise so readily in P3L.

The FAN skeleton framework[5] provides a source notation and an extensible library of transformations for the development of skeletal parallel programs. The source notation is more explicit in its depiction of data dependencies and of parallelism than Adl and has a richer type system. The FAN transformational system is interactive which gives it broader scope, along with more exacting requirements including the need for a cost model[46], than an automated parallelisation system of the type found in the Adl project.

6.6 Conclusions and Future work

This chapter described methods for the parallelisation of point-free BMF code and discussed issues that arise when targetting parallel code to a real distributed architecture.

The effectiveness of the parallelisation process was demonstrated through a detailed simulation of the performance of a number of parallelised example programs. It was shown that a good level of performance is attainable through the application of the parallelisation process.

Early prototype implementations of the paralleliser and code-generator have thus far borne-out the findings of this chapter.

There is scope for incremental improvement of the Adl paralleliser and code-generator. First steps will involve the completing and refining early prototypes for these stages. Other useful developments are in rough order of priority are:

1. Improved memory-management for generated imperative code.

2. The use of a cost-model to assist with some parallelisation decisions.

[46]an interesting observation is that under the FAN cost model, *most* of the transformation rules presented improve programs under all circumstances. Such unconditionally beneficial rules might all be applied automatically prior to the application of conditional rules but, because the transformation system is not confluent, such an approach may preclude the finding of an optimal version for each program.

3. A new fine-tuning stage to incrementally improve parallelised code using conditional transformations coupled with a cost model.

4. The introduction of parallelism within alltup functions.

5. The introduction of nested parallelisation.

6. Facilities for tracing the parallelisation process and the efficiency of code during its development.

7. Allowing more general block-decomposition of nested vector data[47].

The first of these changes has the most potential to immediately improve performance and, thus, has the highest priority. The second and third changes add a fine-tuning stage to the parallelisation process. The fourth change exploits parallelism on the second aggregate data type. The fifth change extends the existing parallelisation process to deeper levels. The sixth change introduces facilities for debugging. The last change may involve the introduction of new datatypes to the source language.

This concludes our discussion on the parallelisation process and the details of the Adl implementation in general. The next chapter summarises and puts into context the findings of this work and points the way to future endeavours within the context of the Adl project.

[47]Perhaps through the agency of a new type for multi-dimensional arrays but possibly through nested parallelism combined with judicious mapping of abstract nodes to hardware nodes.

Chapter 7

Conclusions and Future work

This report described a compilation process which maps a simple functional language, through point-free BMF to distributed parallel code. The first three chapters provided, respectively, an overview of the process, a description of the source language, Adl, and a definition of the point-free BMF dialect used as an intermediate form. The fourth chapter defined a translator from Adl to point-free BMF. The fifth chapter described an effective global optimisation process working via the systematic application of local re-write rules. Finally, the sixth chapter described a parallelisation process and analysed the performance of parallelised code under different assumptions.

The findings relating to this work have been described in these earlier chapters. We provide a summary here.

7.1 Primary findings

It has been shown that the compilation process is effective in rationalising data movement in a number of programs and producing programs exhibiting good speedup on a simulated distributed architecture with reasonable performance parameters[1]. This finding is a proof of concept of the utility of point-free BMF notation and transformations as a medium for optimisation and parallelization.

Another observation, reinforced by this work, is that it is too costly, in a distributed context, to copy every value in scope of a distributed function to each

[1]Early work on a prototype code generator to C-MPI on a cluster[111] indicates that good speedup is achievable on real architectures.

node executing that function. This is widely understood and there are a range of strategies employed by different implementations to rationalise the amount of copying. For example, complementary studies within the Adl project [54] copied only scalar values unconditionally and allowed random access to remote vector elements at run-time, thus transporting only the vector elements actually referenced. Copying can also be reduced by prohibiting references to free variables in functions embedded in skeletons, as is done in EL*[117], or by having the programmer express data distribution requirements in some explicit manner, a strategy used by P3L[31] (among others). Our work reduces the overhead of copying by performing a novel kind of dataflow analysis where a wave of generated code expressing the precise data needs of downstream code is propagated backwards through the program. When analysing references to vector elements it is often impossible to statically determine which elements might need to be copied. In our implementation, such references are integrated into a *distributed-select* operation that performs the random access specified by indexing operations in a consolidated fashion at runtime.

Another contribution of this work is the definition of an *automatic* process to convert implicitly parallel BMF to explicitly parallel BMF via the propagation of a split construct upstream through the code. At this stage, a preliminary prototype implementing this process[146] has been successfully applied to a number of programs.

7.2 Secondary findings

Several secondary observations arise from our work on this implementation.

7.2.1 Incremental transformation offers advantages

The first of these observations is that the methodology of incremental transformation *does* appear to make semantics preservation simple in practice by:

1. allowing semantics-preserving rules to be added, and tested, incrementally and
2. making errors in transformation rules relatively easy to track down, in part due to the above point, but also because the transformation process consists of small transparent steps[2] rather than large opaque ones.

[2]that can be traced in the debugging environment provided by Centaur.

7.2.2 Catch-all rules, if overused can lead to poor performance

A second observation concerns trade-offs occurring through the use of catch-all rules to provide default behaviour for code yet to be handled, during the construction of the optimiser. The primary advantage of catch-all rules is that they allow a partially complete rule-set to be tested on programs containing code not yet explicitly handled by that rule-set. The main drawback is that the lack of processing that a catch-all rule entails creates a potential performance bottleneck from the code it fails to process. If the catch-all rule in question happens to be in a normalising rule-set then the impact of the bottleneck can be amplified by the subsequent application of other rule-sets which, typically, fail to properly optimise the code that has not been properly prepared by the normalising rule-set.

As a general observation, catch-all rules are a valuable mechanism for allowing rule-sets to be constructed and tested incrementally, but the bottlenecks that catch-all introduce can obscure the performance benefits that the other rules in the set might introduce. Said another way, rules can be introduced and tested incrementally but we should not expect to observe most of the improved performance they generate until explicit rules, to handle all code of interest, are in place.

7.2.3 The importance of normalisation

A third observation is the importance of normalisation during the compilation process. The diversity of source-code, coupled with the changes wrought over time by incremental transformations, can result in an unmanageable case explosion. To avoid this problem, we developed rule-sets with the sole purpose of making code more predictable. The normalisation process is assisted by BMF's amenability to transformation. Rule sets that performed normalisation were employed to remove superfluous identity functions, to associate binary function compositions to highlight functions of interest, and to reduce the length of sequences of composed functions (compaction). Armed with the assumptions enforced by normalisation, the task of subsequent rule-sets is much easier. It should be noted that to be effective, normalisation must be applied frequently, at the cost of extra compilation time. For most scientific applications, we believe, that the trade-off between effective optimisation and extra compilation time is worthwhile.

7.2.4 Complex transformation rules are best avoided

A fourth observation, strongly related to the above, is that complex rules, designed to handle diverse code, are best avoided. Complex rules are more difficult to verify and maintain than simple ones. We found that it is, generally, much easier to use normalisation to reduce the diversity of code and thus reduce the need for complex rules to handle that code[3].

7.2.5 Observations relating to Centaur

Our prototype implementation is one of the larger scale applications of the Centaur system[24]. The Centaur system is an integrated prototyping environment specialised for defining language processing components using Natural semantics. Very substantial support is also provided for parsing, debugging and pretty printing. We used Centaur very heavily, using it to implement the parser, several pretty-printers, the translator and the optimiser for our prototype. In early stages of the project we also used Centaur to implement a simple typechecker and interpreter. We have gained considerable experience in using this system and some observations follow.

Natural Semantics is a good medium for describing compilation Natural semantics allows a process to be described using sets of inference rules working over objects of defined type. Such rules provided a good substrate for expressing our processes quickly and succinctly. Once ideas are expressed in these rules Centaur, with little extra effort, can turn them into a working prototype.

Some compromise is required to produce an efficient prototype The simplest way to express ideas in Natural Semantics, and in almost any other programming medium for that matter, is to write rules in a purely declarative manner with little regard for efficiency. Unfortunately, we quickly found that overheads from backtracking were such that little useful computation could be done without compromising the structure of the rules for the sake of efficiency. Our final prototype is tolerably efficient but some of the original elegance of the rules is gone.

[3]The application of this principle of: *change the code not the rules* is made easier to implement by the innate transformability of BMF code.

Centaur can take a very long time to fail The logic engine underpinning Centaur's implementation of Natural Semantics is Prolog. As with any Prolog implementation this engine is relentless in pursuit of potential solutions even when no solution exists. Small bugs in source code or in rule-sets often lead to the system spending large amounts of time engaged in backtracking. To avoid this problem we introduced more guards into our rules to prevent them being spuriously applied and occasionally employed a non-logical construct *Once* to prevent backtracking by allowing only one attempt at the proof goal forming its argument.

The Centaur environment is very useful The infrastructure provided by Centaur was very useful in this project. In particular, we used the semantic debugger very heavily. We also made heavy use of the pretty-printing facilities for both viewing programs on-screen and for dumping concrete syntax to files for further processing/analysis.

7.3 Future work

There is much scope for future work. Of the highest priority is the development of a phase of code generation to optimise memory management. Such a phase will, where possible, utilise existing techniques to avoid allocation and copying such as those used in the SISAL[56, 55] and FP*[144, 145] compilers.

New features can be added to the Adl language to including the introduction of true multi-dimensional arrays to provide more flexibility in partitioning data and some support for recursion can be added to provide a greater level of expressive power.

On the implementation side, there is scope to provide support for nested parallelism. There is also potential to add instrumentation to the optimisation and parallelisation processes to provide a trace of its progress. Such a trace can be combined with a static cost model to measure the impact of optimisation and parallelisation at various stages.

There is also scope to explore the targetting of point-free code to alternative platforms. In this work, our focus was on tightly coupled distributed memory architectures. Emerging on-chip parallel platforms such as Explicit Data Graph Execution (EDGE) architectures[26, 133], a-priori, offer opportunity for exploiting parallelism in point-free notation at a fine-grained level.

Finally, the compilation methodology used in this work offers great potential. The systematic use of local rewrite rules to propagate changes of global scope provides advantages in terms of flexibility and transparency. The use of normalisation is key to the effectiveness of this process. There is scope for improving and extending the definitions of our compilation process. First, by the reuse of rules can be enhanced by the separation of rewrite rules from the strategies for their application. Systems such as Stratego[139] support such separation. Second, by the use of more formally defined syntactic interfaces between transformation components, in a similar manner to[138]. The use of interfaces in particular offers the prospect of enhancing a compiler by simplifying the incremental addition of new modules.

In summary, the implementation described in this work has demonstrated good performance in its application domain. We have only just started to explore the potential of point-free form as a medium for compilation, and to develop the techniques necessary to exploit this potential. There remains much interesting work to be done.

Appendix A

Glossary

This report defined a number of new terms and used, in a specialised context, a number of existing terms. This appendix lists new terms and puts others into context. All terms are listed in alphabetical order.

address function: Any BMF function that references an element of a tuple. All address functions are written ${}^m\pi_n$ where m is the arity of the input tuple and n is the element of the tuple being referenced.

aggregate function: Any function over an aggregate type or returning an aggregate type. map is an example.

aggregate type: A vector or a tuple type.

alltuple: A second order function that applies a tuple of functions to copies of an input value, returning a tuple of results. Sometimes called alltup or written $(\ldots)^\circ$.

allvec: A second order function that applies a vector of functions to copies of an input value, returning a vector of results. Usually written $[\ldots]^\circ$.

array: See **vector**.

associative: Property of a binary function. A binary function $\oplus$ is associative if:

$$\forall x, y, z : (x \oplus y) \oplus z = x \oplus (y \oplus z)$$

All binary functions given to parallelisable versions of `reduce` and `scan` must be associative in order to guarantee deterministic behaviour.

axiom: A rule consisting of an assertion with no premises. An example of an axiom is $f \cdot \mathsf{id} \Rightarrow f$.

backtracking: Attempting an alternative path to proof upon failure of the current rule. Backtracking is, potentially, a major source of inefficiency in the Adl prototype implementation. Rules are carefully crafted to avoid redundant backtracking with some compromise to the declarative style of the semantics.

barrier synchronisation: A synchronisation event in which every member of a group of processes participates. Very good for simplifying the state space of a parallel application. In the Adl project the composition of parallel components is currently implemented as a barrier synchronisation.

binary function: In the Adl project, any function accepting a pair of values.

BMF (language): A point-free subset of Bird-Meertens-Formalism used as an intermediate language in this work.

BMF (theories): The notation and algebra of Bird-Meerten's formalism as applied to different aggregate types. There is a BMF theory for each of a number of types.

body: The core part of a function or expression. For example, the body of a map function is the function that it encapsulates. The term body is necessary to differentiate the code that performs the core activity of a function from the code that is used to support that activity.

compaction: A normalisation process using rewrites to reduce the number of function compositions between the function of interest and the current site of optimisation.

constant function: A function that, that when applied to any value, yields a constant.

category theory: The abstract study of objects and arrows. The work upon which this implementation is, primarily, derived from the category **Set** of sets and total functions. Several BMF identities on concatenate lists have proofs, at least partly, reliant on category theory.

Centaur: A language system which takes a syntax description, a semantic description and a pretty printer description and produces a working prototype of a language or process.

concatenate-list (cf. vector): A concatenate list is an aggregate data type defined by the constructors $+\!\!+$ (concatenate), $[\cdot]$ (make-singleton) and $[]$ (the empty list). Because $+\!\!+$ is associative such lists can be decomposed in arbitrary orders, which allows parallelism. The constructors above are not used directly by the Adl programmer. Other functions, such as `iota` are used to provide their functionality in an indirect way.

conclusion/rule conclusion: The part of an inference rule that is below the line.

code-request: The newly optimised code produced by an optimisation rule. Typically, optimisation rules output a code-request which is optimised and a wish-list which refers to temporary code forming a bridge between the code-request and unoptimised code upstream of the wish-list.

composition: Function composition, denoted: "$\cdot$". A binary function to combine function such that the output of the right-hand function becomes the input of the left-hand function. That is: $f \cdot g\, x = f(g\, x)$

denotational: Denoting a value. A definition concerned only with a value.

distl: Distribute-left. A function, taking a pair consisting of a value and a list, that distributes the value over the list to form a list of pairs. Can be implemented using zip and repeat.

distributed memory architecture: Any computer hardware where memory is strongly associated with a particular processor and the cost-difference between access to the memory associated with a processor and access to other, remote, memory is relatively large.

Data Movement Optimiser: The part of the Adl implementation responsible for statically reducing the cost associated with data movement in the BMF intermediate form.

downstream: The direction in which data moves in a BMF program. Because programs are expressed in point-free form in this report. Downstream is synonymous with leftwards.

environment: In the context of the translator, the term *environment* refers to the mapping between variables and the BMF functions used to access their values. In a more general context, the term environment refers to the variables that are free in the scope of the original Adl function. In point-free BMF, all values that are used by a function must be explicitly passed to that function so the environment is tupled with the original input parameters.

expression: A term returning a value.

fail: The state that occurs when a premise of a rule cannot be proven. A fail will result in the inference engine backtracking to find a new rule.

filter (function): A common second-order function to eliminate elements of a input list according to an input predicate. Not currently included as a primitive in the Adl implementation.

filter (optimisation): A data-structure, used in tuple-optimisation, whose values can be either BMF code or null. A filter expresses the data needs of downstream code. The null value is needed to express the case of downstream code having no specific data needs. In the current implementation a null filter is generated only by constant functions.

front-of-optimisation: see optimisation-front.

function (partial): A function that has undefined elements in its range for some elements in its domain. An example of a partial function is div.

index function/operator: (!) a function for randomly accessing a vector at a given location.

index generating function: The code generating an integer forming the second argument to an index operator.

indexing function: An index operator composed with the alltup function generating its input.

init: A function that returns all but the last element of its input vector.

id: The identity function. The function that always returns its input.

identity (algebraic): An equation for rewriting expressions, in general, and programs consisting of composed functions, in the context of this work.

implementation (noun): A compiler.

instance: A single invocation of a portion of code, e.g. a `map` body.

iota: A primitive function that dynamically allocates a list containing a sequence of successive integers, starting at zero. Used as the core means for dynamic allocation of storage in Adl.

lambda lifting: Making free-variables bound. A necessary part of Adl to BMF translation.

last: A function returning the last element of a vector.

map: A second order function for applying an input function, elementwise to each element of a vector. `map` is a primitive in Adl.

map body: The functional parameter of the `map` function.

map-translation: The BMF code, produced by the translator, corresponding to a `map` function in Adl.

mask: Function to filter a vector according to a boolean mask vector of the same length.

model: In the context of this work: a model of the evaluation and cost of running BMF programs. The most detailed model used in this work is a trace-generating simulator that executes parallelised BMF code on a simulated distributed machine.

Natural Semantics: The semantic framework used to express processes in the Centaur system. Most parts of the Adl implementation were defined using Natural Semantics. Natural semantics shares many features with Structured Operational Semantics.

nested vector: A vector of vectors. Vectors in Adl have only one dimension so a nested vector is the only way of approximating multidimensional structures. There is no requirement that the sub-vectors all have the same length.

normalisation: The process of making code predictable for the consumption of later rules. In the Adl implementation, examples of normalisation include, removing identity functions, compaction, and associating functions to the left or right.

optimisation front: The site in the BMF code where optimisation is currently taking place. This point moves from downstream to upstream as optimisation progresses. At the optimisation front a wish-list is optimised w.r.t the upstream code to form a code-request and a new-wish list which is now further upstream.

pair: A tuple with two elements.

parallelisation: The process of converting BMF code which is implicitly parallel into explicitly distributed BMF code. This process is defined in a similar way to the optimisation process with parallelisation starting from the downstream parts of the program and moving upstream.

π_1, π_2 **etc:** Functions to project elements from tuples. π_1 projects the first element from a tuple, π_2 the second and so on. When the arity of the input tuple is more than two, the arity of the input tuple is written as a prefixed superscript. For example ${}^3\pi_3$.

permutation: A re-ordering of a data structure.

polymorphic: The property of being able to handle more than one type. `map`, `reduce` and `scan` are polymorphic functions. Adl allows a limited amount of user-defined polymorphism through limited overloading of user-defined functions.

point-free: A style of programming without the use of named variables or parameters. Using point-free style, functions are glued together with second-order operations such as function composition and alltup.

predicate: A function returning a boolean value.

prefix(parallel): A parallel variant of `scan`.

premise: A clause above the line of an inference rule. All premises must be proved true before the rule is proved true.

priffle: Predicate-riffle. A function to merge elements of two sub-vectors into a single vector with the guidance of a vector of boolean values specifying which vector to extract the next value from.

program(BMF): A BMF function intended for execution.

product type: A tuple.

proof: A sequence of inference steps ascertaining the truth of a desired goal. In the context of the Adl project the translator, optimiser and (to a lesser extent) the paralleliser are executable proof processes.

ragged vector: A vector of vectors of varying length.

range generator: The code in a translated `map` function that generates the vector of values corresponding the second input value to the original Adl `map` function.

reduce: A primitive function to insert a binary operator between the elements of a vector, evaluate this expression and return a single result. `reduce` has a highly parallel implementation.

reduce body: The code appearing on the left-hand-side of the / symbol in the translation of a `reduce` function.

repeat: A BMF function that takes a pair of the form (a, n) and produces a vector containing a repeated n times. A distributed version of `repeat` is used to broadcast values.

rule-set: A named group of inference rules, written in natural semantics with a well defined purpose.

`scan`: A primitive function that takes a binary operator and a vector and returns a vector containing the results of inserting the operator between the elements of each of the initial segments of the vector. As with `reduce`, a highly parallel implementation exists for `scan`.

scan body: The code appearing on the left-hand-side of the // symbol in the translation of a `scan` function.

select: A BMF function that takes a pair of vectors (s, t) and returns the elements of s indexed by the values in t.

split: A BMF function that takes a vector and partitions it into a vector of distributed vectors.

strict: An evaluation order were arguments to functions are evaluated prior to being passed into the body of the function. Adl is strictly evaluated.

surface: That part of a BMF function that has no other functions composed upstream of it.

syntactic polymorphism: Polymorphism implemented by instantiation of polymorphic functions to the types of their actual parameters. With this type of polymorphism, a function can be instantiated more than once when it is called with different types.

translator: The phase of the Adl compiler responsible for converting Adl code to BMF code.

translator code: The code produced by the Adl to BMF translator.

transpose: In the context of this work, a primitive BMF function to re-arrange the elements of two dimensions of a nested vector so that indices to access each element are swapped in those dimensions. When applied to ragged vectors transpose may mix undefined elements into its result.

tuple: A product-type containing two or more values of possibly different type.

tuple-optimisation: Pass of the optimiser responsible for reducing the quantity of data transported to support access to elements of tuples.

Typol: The language used to express the natural semantics definitions executed by Centaur.

unary operator: A function of one non-tuple parameter.

upstream: The direction from which data flows in BMF programs. In the examples in this report, upstream is a synonym for rightwards.

vector: An array structure restricted to one-dimension. Vectors can be nested.

vector-optimisation: Pass of the optimiser responsible for reducing the quantity of data transported to support access to elements of vectors.

vector referencing function: A function generating a vector forming the first argument of an index operator.

wish list: A section of BMF code forming a bridge between already-optimised code downstream and yet-to-be optimised code upstream. Typically, the wish-list extracts a small amount of output data from a large amount of input data.

zip: A function that takes a pair of vectors and produces a list of pairs from corresponding elements of the input vectors.

Appendix B

The transpose function

This chapter has three parts. The first part describes the issues surrounding the implementation of parametric transpose on nested vectors. The second part is an annotated listing of a Scheme program implementing parametric transpose. The third part is a transcript of the output of the model run against selected test data.

B.1 Implementation of parametric transpose

What is the meaning of a parametric transpose operation? An informal definition is:

$$(\ldots(\ldots(\ldots(\mathsf{transpose}_{(x,y)}\ v)\ldots!i_x)\ldots)!i_y\ldots) = (\ldots(\ldots(v\ldots!i_y)\ldots!i_x)\ldots)$$

for all indices i_x valid in the xth from-outer dimension of v and all indices i_y valid in the yth-from-outer dimension of v and where all other index values are preserved across the equals.

Figure 170 gives a concrete illustration of the effect of transpose of dimension 0 (outermost) and dimension 2 (in this case, innermost) of a triple-nested vector. Note, that any practical implementation of transpose has to decide which indices to iterate over in order to build the transposed vector. The answer is, generally, that the index that each dimension of the transposed vector should range over is the length of corresponding dimension of the original vector. Also note that, even though the transposed dimensions x and y have moved they should still range over the lengths they had in the original vector. This preservation of range means that when the length of x and the length of y are different, the shape of the transposed vector is different.

$$\mathsf{transpose}_{(0,2)} \begin{array}{l}[[[1,2],[3,4],[5,6]],\\ \;[[7,8],[9,10],[11,12]],\\ \;[[13,14],[15,16],[17,18]],\\ \;[[19,20],[21,22],[23,24]]]\end{array} = \begin{array}{l}[[[1,7,13,19],[3,9,15,21],[5,11,17,23]]\\ \;[[2,8,14,20],[4,10,16,22],[6,12,18,24]]]\end{array}$$

Figure 170. An illustration of the effect of $\mathsf{transpose}_{(0,2)}$ on a triple-nested vector. Note that the length of the middle dimension, the one unaffected by this operation is untouched while the lengths of the other two dimensions are exchanged.

$$\mathsf{transpose}_{(0,1)} \cdot \mathsf{transpose}_{(0,1)} \begin{array}{l}[[1,2],\\ \;[3,4,5],\\ \;[6],\\ \;[7,8,9,10]]\end{array} \neq \mathsf{transpose}_{(0,1)} \begin{array}{l}[[1,3,6,7],\\ \;[2,4,8],\\ \;[5,9],\\ \;[10]]\end{array} \neq \begin{array}{l}[[1,2,5,10],\\ \;[3,4,9],\\ \;[6,8],\\ \;[7]]\end{array}$$

Figure 171. An illustration of how naive transposition of a jagged vector leads incorrect results. Using well-behaved transposition the last vector would be the same as the original input vector.

There is one minor shortcoming in the foregoing analysis. We have referred to the *length* of a dimension as if such a thing exists. Unfortunately, when the subvectors of one or more dimensions have different lengths there is no such thing as a single length for that dimension. How such *jagged* vectors are handled is discussed next.

B.1.1 Transposing non-rectangular vectors

transpose works well with rectangular vectors and, we envisage, vectors that require transposition will, almost invariably, be rectangular. However, a complete implementation will need to work with all semantically-correct programs and it is possible, using carefully chosen index values, for a program to be both semantically-correct and specify transposition of a jagged vector.

Transposing jagged vectors is a non-trivial operation. Figure 171 shows how easily the process goes wrong. A basic identity that we expect to hold for transpose is:

$$\mathsf{transpose}_{(0,1)} \cdot \mathsf{transpose}_{(0,1)}\ x = x$$

That is, if we transpose the same dimensions of a vector twice we expect to get back the same vector. In figure 171 this is most definitely *not* the case.

$$\begin{array}{cccccccc}
 & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \\
\cdots & \perp & \perp & \perp & \perp & \perp & \perp & \cdots \\
\cdots & \perp & 1 & 2 & \perp & \perp & \perp & \cdots \\
\cdots & \perp & 3 & 4 & 5 & \perp & \perp & \cdots \\
\cdots & \perp & 6 & \perp & \perp & \perp & \perp & \cdots \\
\cdots & \perp & 7 & 8 & 9 & 10 & \perp & \cdots \\
\cdots & \perp & \perp & \perp & \perp & \perp & \perp & \cdots \\
 & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots &
\end{array}$$

Figure 172. A nested vector floating in a sea of undefined values. Any attempt to access beyond the bounds of a vector returns the undefined value $\perp$.

$$\begin{aligned}
& \mathsf{transpose}_{(0,1)} \cdot \mathsf{transpose}_{(0,1)} \begin{array}{cccc} 1 & 2 & \perp & \perp \\ 3 & 4 & 5 & \perp \\ 6 & \perp & \perp & \perp \\ 7 & 8 & 9 & 10 \end{array} \\
= \; & \mathsf{transpose}_{(0,1)} \begin{array}{cccc} 1 & 3 & 6 & 7 \\ 2 & 4 & \perp & 8 \\ \perp & 5 & \perp & 9 \\ \perp & \perp & \perp & 10 \end{array} \\
= \; & \begin{array}{cccc} 1 & 2 & \perp & \perp \\ 3 & 4 & 5 & \perp \\ 6 & \perp & \perp & \perp \\ 7 & 8 & 9 & 10 \end{array}
\end{aligned}$$

Figure 173. An illustration of how padding, provided by $\perp$ values, leads to the correct behaviour of **transpose**.

The problem is caused by misaligned values and a cure for the problem is to take account of the space surrounding the jagged vector when performing our transposition. Our approach is to imagine the jagged vector as floating in a sea of undefined values. Figure 172 illustrates this view. Any attempt to reference values beyond the bounds of the vector returns $\perp$ (pronounced "bottom"), the undefined value. Following the convention used by FP*[145] it is the programmer's repsonsibility to make sure that programs do not access undefined values. The $\perp$ values, within the bounding box surrounding the defined values of the jagged vector, provide valuable padding for the transpose function. Figure 173 shows how transposition of the entire bounding box enclosing a jagged vector behaves correctly. Note that the middle vector, in particular, is interspersed with $\perp$ values. These values correspond to attempting to access values beyond the bounds of the original nested vector[1].

Using $\perp$ comes at a cost. Values in a transposed vector can be either, a valid value or $\perp$. This representation requires a tagged-union type with the run-time penalty that this representation entails. However, $\perp$ will be accessed only in incorrect programs and there is a strong argument that an implementation should at least allow the use of the tagged-union type to be switched off once the programmer is confident that the program is working.

B.2 Source code of an implementation

The source code for a test-implementation of parametric transpose, written in MIT-scheme, follows.

An electronic copy of the source can be downloaded from:

```
http://www.cs.adelaide.edu.au/~brad/src/Adl/scheme/transpose.scm
```

The basic method used by the implementation is to convert an arbitrarily nested list[2] into a flattened bag of data/index-list pairs. The indices of the bag are manipulated by transpose and the bag can then be converted back into a list.

An intuitive understanding of the process can be had by loading the source code shown in the next section into the scheme interpreter and executing the following sequence of evaluations.

```
list1
simple-transpose-list1
list2
jagged-transpose-list2
list3
ident-list3
backw-ident-list3
```

[1]From experiments with an experimental implementation of transpose written in scheme it is apparent that two versions of $\perp$ are needed. One version is required for attempts to access beyond the bounds of a sub-vector. Another less common version $\perp_e$ is required to denote an attempt to access the first element of an empty vector. At first glance this might not seem like a special case but the special form is required so that the empty vector can be reinstated if/when the vector containing the empty vector is returned to its original orientation.

[2]Scheme's relaxed typing allows such arbitrary nesting without defining extra types.

Note that the symbols `%` and `e%` are place-holders for references beyond the bounds of the original vector and references to the first element of an empty vector respectively. These place-holders are vital if transpositions are to be reversible in general. A transcript of the output of the above expressions appears in the last section of this chapter.

B.3 Source code for `transpose.scm`

```
;;;; Model for sequential transpose
;;;; ------------------------------
;;;; Works by converting a nested list
;;;  to a bag.
;;;; The bag elements are then manipulated
;;;; Finally, the bag is turned back into a nested
;;;; list.


(define (list-to-bag l) (lb-iter l 0 ()))

(define  (lb-iter l c v)
  (cond ((null? l)  ())
        ((pair? l) (append
                        (if (null? (car l))  ;;; the first element is empty list?
                            (list (list () (append v (list c))))
                            (lb-iter (car l) 0 (append v (list c))))
                        (lb-iter (cdr l) (+ c 1) v)))
        (else   (list (list  l v))))) ;;; atom? just write bag entry.


;;; Auxiliary functions
;;; different-outers compares two vectors (not always of
;;; the same length and returns true if the outer dims
;;; are different. Looks at all but the last elements of
;;; the longest vector.

(define (different-outers c t)
  (cond ((null? c) #f)
        ((or (and (= 0 (car c)) (eq? '% (car t)))
             (= (car c) (car t)))
         (different-outers (cdr c) (cdr t)))
        ((= 1 (length c)) #f)
        ( else   #t)))

;;; length, function to find the length of a list.

(define (length l)
  (if (null? l)
      0
      (+ 1 (length (cdr l)))))


;;; different-inner compares the innermost elements (last
;;; elements of the current-index vector and target-index
;;; vector. If the end of the current index vector is
;;; different from the same element of the target-index
;;; vector then there is, possibly, a missing element.

(define (different-inners c t)
  (cond ((or (null? c) (null? t)) #f)
        ((= 1 (length c)) (not (= (car c) (car t))))
        (else (different-inners (cdr c) (cdr t)))))
```

```
;; current-shorter returns true if the current vector is shorter
;; than the target vector. If it is (and it is the same otherwise)
;; then we should add a zero element to the end of the current-index

(define (current-shorter c t)
  (cond ((null? t) #f)
        ((null? c) #t)
        (else (current-shorter (cdr c) (cdr t)))))


;; append-zero returns the input list with a zero appended to the
;; end. This is used when the current vector is shorter than the
;; target value but identical otherwise.

(define (append-zero c)
  (append c (list 0)))

;; increment the number associated with the last element of
;; the input list.

(define (inc-last l)
  (cond ((null? l) ())
    ((= 1 (length l)) (list (+ 1 (car l))))
    (else (cons (car l) (inc-last (cdr l))))))

;;; main-function. Calls bag-to-list-iter and then extracts
;;; the first part of the state. The second part of the state
;;; should be the empty target value list -- if all works well.

(define (bag-to-list l)
  (if (null? l)
      ()
      (car (bag-to-list-iter (list 0) l ))))
```

```
;;; The function that does all the work.
;;;

(define (bag-to-list-iter c l)
  (if
   (null? l)
   (cons () ())
   (let ((t (car (cdr (car l))))
         (v (car (car l))))
      (if (different-outers c t)
          (cons () l)
          (if (different-inners c t)
              (let ((state1 (bag-to-list-iter (inc-last c) l)))
                (cons (cons '% (car state1)) (cdr state1)))
              (if (current-shorter c t)
                  (let* ((state1 (bag-to-list-iter (append-zero c) l))
                         (state2 (bag-to-list-iter (inc-last c) (cdr state1))))
                    (cons (cons (car state1) (car state2))
                          (cdr state2)))
                  (let ((state1 (bag-to-list-iter (inc-last c) (cdr l))))
                        (cons (cons v (car state1))
                              (cdr state1)))))))))



;;;; The transpose operations (and its auxiliary defs) follow.
;;;; transpose is broken into four stages.
;;;; 1. conditional-pad: scans the bag and pads the arrays out to
;;;;    the highest swapped dimension. Any entry needing pading
;;;;    will have its value component changed to '%e
;;;; 2. swap-dims: will swap the elements in the dimensions of
;;;;    each index.
;;;; 3. unpad: will remove trailing padding from the elements.
;;;; 4. restore-empty: will replace '%e's with () if there are
```

```
;;;;    no # values in the corresponding target index lists.


;;; conditional pad - scan the bag and pad out the arrays to the
;;; highest dimension in the dimension pair argument.

(define (conditional-pad ds b)
  (letrec ((mds (max (car ds) (cdr ds)))
           (cond-pad (lambda (b)
                       (if (null? b)
                           ()
                           (cons (pad-one mds (car b))
                              (cond-pad (cdr b)))))))
    (cond-pad b)))


;;; pad-one: check the length. If it is greater than the index, do nothing.
;;; if it is less than or equal to the index then pad the target index list
;;; out with the difference.

(define (pad-one m be )
  (let*
      ((t (car (cdr be)))
       (l (length t))
       (gap (- m (- l 1))))
    (if (> gap 0)
        (list '%e (append t (repeat gap '%)))
        be)))

(define (repeat n item)
  (if (= n 0)
      ()
      (cons item (repeat (- n 1) item))))
```

```
;;; Swap dimensions in  each element of the bag-list
;;; the second stage of transpose.
(define (swap-dims d1 d2 l)
  (if (null? l)
      ()
      (cons (list (car (car l))
                  (swap-one d1 d2 (car (cdr (car l)))))
            (swap-dims d1 d2 (cdr l)))))



;;; Swap one

(define (swap-one f s l)
  (let*
      ((fst_val (index l f))
       (snd_val (index l s))
       (fst_tmp (insert l f snd_val))
       (snd_tmp (insert fst_tmp s fst_val)))
    snd_tmp))

;;; index operation. returns the (n-1)th element of l
(define (index l n)
  (if (= n 0)
      (car l)
      (index (cdr l) (- n 1))))

;;; insert operation. returns l with the nth element changed to v

(define (insert l n v)
  (if (= n 0)
      (cons v (cdr l))
      (cons (car l) (insert (cdr l) (- n 1) v))))

;;; remove-excess-padding
;;; traverse the list. For each target remove excess #'s at the end of
;;; the list.

(define (remove-excess-padding l)
  (if (null? l)
      ()
      (cons (remove-one-lot (car l))
            (remove-excess-padding (cdr l)))))

(define (remove-one-lot be)
  (list  (car be) (remove-from-target (car (cdr be)))))

(define (remove-from-target t)
  (let*
      ((rt  (reverse t))
       (crt (remove-leading-errors rt))
       (ct  (reverse crt)))
    ct))


;;; generic reverse function

(define (reverse l)
  (letrec
      ((rev-iter (lambda (i r)
                   (if
                    (null? i)
                    r
                    (rev-iter (cdr i)
                              (cons (car i) r))))))
    (rev-iter l ())))

;;; function to remove leading '% elements
```

```
(define (remove-leading-errors t)
  (cond ((null? t) ())
        ((eq? '% (car t)) (remove-leading-errors (cdr t)))
        (else t)))



;;; revert empty.. produces an empty list if the coordinates of a '#e
;;; element are all valid.
;;; function is

(define (revert-empty l)
  (map revert-one l))

(define (revert-one be)
  (if (eq? (car be) '%e)
      (check-valid-coord be)
      be))

(define (check-valid-coord be)
  (if (valid-coord (car (cdr be)))
      (list () (car (cdr be)))
      be))

(define (valid-coord t)
  (cond ((null? t) '#t)
        ((eq? (car t) '%) '#f)
        (else (valid-coord (cdr t)))))



;;; now we need to define a sorting function
;;; to return the bag to a form that is easy to
;;; convert back to a list.

;;; use quicksort
```

```
(define (filter f xs)
  (cond ((null? xs) ())
        ((f (car xs)) (cons (car xs) (filter f (cdr xs))))
        (else (filter f (cdr xs)))))


;;; The quicksort function
;;; issues recursive calls relies on a binary  filter predicate p.
;;; this p will be the equivalent of ">" (i.e. true if first arg is
;;; ">" the second.

(define (quicksort l p)
  (cond
   ((null? l) ())
   ((= 1 (length l)) l)
   (else
    (let ((left  (filter (lambda (x) (p (car l) x)) (cdr l)))
          (right (filter (lambda (x) (not (p (car l) x))) (cdr l))))
      (append (quicksort left p)
              (append (list (car l))
                      (quicksort right p)))))))


;;; custom binary comparator operator for bag elements.
;;; the items are compared on the target index lists (second component)
;;; rather than on their value component. Remember # = 0 for our
;;; purposes.

(define (bag-index-greater-than x y)
  (define xt  (car (cdr x)))
  (define yt  (car (cdr y)))
  (define turn-zero (lambda (n) (if (eq? '% n) 0 n)))
  (define xtn (map turn-zero xt))
  (define ytn (map turn-zero yt))
  (define gt-iter (lambda (xs ys)
  (cond ((null? xs) '#f) ;; must ask this question before next
((null? ys) '#t) ;; must be asked after last.
((= (car xs) (car ys)) (gt-iter (cdr xs) (cdr ys)))
((> (car xs) (car ys)) '#t)
(else '#f))))
    (gt-iter xtn ytn))

;;; now for the transpose operator
;;; it takes a bag and puts it through 4 stages.
;;; first it takes the bag and pads out all of the
;;; index lists that have lengths shorter than the highest
;;; transposed dimension. This will occur with empty lists
;;; (whose index lists can be shorter than those items which
;;; need all dimensions in the array to specify their location.
;;; The second step is the index swap for the transpose itself.
;;; This is a straightforward step.
;;; Some transpositions can leave trailing invalid index elements
;;; in the index lists. These can be removed with no effect on future
;;; operations (the semantics says that there are infinitely many of
;;; these trailing valuesanyway). These can be
;;; eliminated by remove-excess-padding.
;;; Finally, the resulting bag is sorted using quicksort.

(define (transpose ds b)
  (let*
      ((pb (conditional-pad ds b))
       (spb (swap-dims (car ds) (cdr ds) pb))
       (sb (remove-excess-padding spb))
       (sbr (revert-empty sb))
       (ssb (quicksort sbr bag-index-greater-than)))
    ssb))
```

```
;;; Test data
;;;


(define list1 (list (list 1 2 3)
                    (list 4 5 6)
                    (list 7 8 9)))

(define list2 (list (list 1 2)
                    (list 3 4 5 6)
                    ()
                    (list 7)
                    (list 8 9 10)))

(define list3 (list (list (list 1 2 3) (list 4 5 6) (list 7))
                    (list (list 8 9) () )
                    (list (list 10 11 12) (list 13) (list 14))
                    ()
                    (list (list 15) (list 16 17 18 19))))


;;; Test applications

(define simple-transpose-list1 (bag-to-list
                                 (transpose (cons 0 1)
                                            (list-to-bag list1))))

(define jagged-transpose-list2 (bag-to-list
                                 (transpose (cons 0 1)
                                            (list-to-bag list2))))
```

```
(define ident-list3 (bag-to-list (transpose
                                   (cons 0 1)
                                   (transpose
                                    (cons 1 2)
                                    (transpose
                                     (cons 0 1)
                                     (transpose
                                      (cons 0 2)
                                      (list-to-bag list3)))))))
```

```
(define backw-ident-list3
  (bag-to-list (transpose
                (cons 0 2)
                (transpose
                 (cons 0 1)
                 (transpose
                  (cons 1 2)
                  (transpose
                   (cons 0 1)
                   (list-to-bag list3)))))))
```

B.4 Test results

To test the transpose implementation above we ran the code against a number of test expressions. To illustrate the effect of the transpose operator we show the output of the following expressions at the end of the code above:

`simple-transpose-list1`: runs a single transpose on a rectangular vector, nested to two dimensions.

`jagged-transpose-list2`: runs a single transpose on a jagged vector, nested to two dimensions.

ident-list3: a composed series of transpositions over various dimensions of a jagged vector, nested to three dimensions. The series of transpositions is equivalent to an identity operation on an input vector.

backw-ident-list3: the series of transpositions in **ident-list3** applied to the same input list in reverse.

The transcript of the output of the evaluation of these expressions follows:

```
1 ]=> list1

;Value 5: ((1 2 3) (4 5 6) (7 8 9))

1 ]=>  simple-transpose-list1

;Value 2: ((1 4 7) (2 5 8) (3 6 9))

1 ]=> list2

;Value 6: ((1 2) (3 4 5 6) () (7) (8 9 10))

1 ]=> jagged-transpose-list2

;Value 3: ((1 3 %e 7 8) (2 4 % % 9) (% 5 % % 10) (% 6))

1 ]=> list3

;Value 7: (((1 2 3) (4 5 6) (7)) ((8 9) ())
           ((10 11 12) (13) (14)) () ((15) (16 17 18 19)))

1 ]=> ident-list3

;Value 4: (((1 2 3) (4 5 6) (7)) ((8 9) ())
           ((10 11 12) (13) (14)) () ((15) (16 17 18 19)))

1 ]=> backw-ident-list3
```

```
;Value 1: (((1 2 3) (4 5 6) (7))
           ((8 9) ()) ((10 11 12) (13) (14)) () ((15) (16 17 18 19)))
```

Appendix C

Sorted and Sortable indexing functions

A sorted indexing function is a nested indexing function where the existing rules of the optimiser are able to remove all index operators (!). This chapter precisely defines the property of a nested indexing function, in a map-body, being *sorted.* We also describe the properties that have to hold for an indexing function to be *sortable.*

The following section lists the assumptions and notation employed by the rest of this chapter. Section C.2 defines the set of valid addresses that are amenable to sorting. Section C.3 defines a partial ordering on this set of addresses. Section C.4 defines a process for converting an index function into a list of addresses. Section C.5 uses the definitions in the previous sections to define a predicate to test if an index function is sorted. Finally, section C.6 outlines the process by which an unsorted index expression can be converted to a sorted index expression if it is found to be sortable.

C.1 Assumptions

In what follows we assume that all nested indexing functions take the form:

$$! \cdot (! \cdot (\ldots (! \cdot (v, a_{n-1})^{\circ}, \ldots)^{\circ}, a_1)^{\circ}, a_0)^{\circ}$$

We also assume that each index-generator, a_i:

- has undergone code-compaction, as described on page 102.

- and has had all superfluous identity functions removed with the exception of any $(\pi_1, \pi_2)^\circ$ functions immediately upstream of a primitive non-addressing function[1].

Whether such an indexing function is sorted to a form where it is possible to optimise-away all index operations depends on which addressing functions appear in the index-generating functions $a_0, a_1, \ldots, a_{n-1}$. To extract, and analyse, these addressing functions, we will define:

- a set called *Addr* that contains all valid addressing functions,
- a partial ordering on the members of *Addr*,
- a function called *ExtractAddrs* that extracts the set of all address functions from an index generator,
- a function called *ToAddr* which condenses the output set of *ExtractAddrs* to a single value,
- the property an indexing function being sorted, in terms of the above entities, and
- the property of an indexing function being sortable.

We start by defining *Addr*.

C.2 The set of valid address functions

We define a set *Addr*:

$$Addr = \{p \in B\_EXP | is\_trans\_addr(p)\} \cup \{\mathsf{K}\} \cup \{\bot\}$$

which contains all address functions that can appear in a nested index function along with an additional element K used as a placeholder to denote constant functions which are not dependent on any input value. The undefined value $\bot$ is included to denote

[1]This exception is to handle the cases where the code such as $! \cdot (\pi_1, \pi_2)^\circ$ appears inside code to generate an index value to a nested index function. If the, otherwise superfluous, $(\pi_1, \pi_2)^\circ$ were to be removed it would introduce a special case making detection of code that is amenable to optimisation more difficult.

error values generated by a *ToAddr* function to be defined shortly. The predicate $is\_trans\_addr(f)$ is true iff:

$$\begin{aligned}&(f = \pi_2)\vee\\&(f = op \cdot \pi_2)\vee\\&(f = g \cdot \pi_1) \wedge is\_trans\_addr(g)\end{aligned}$$

Informally, *Addr* is the set:

$$\{\pi_2, op \cdot \pi_2, \pi_2 \cdot \pi_1, op \cdot \pi_2 \cdot \pi_1, \pi_2 \cdot \pi_1 \cdot \pi_1, op \cdot \pi_2 \cdot \pi_1 \cdot \pi_1 \ldots, \mathsf{K}\}$$

where op is any element of $B\_EXP - \{\pi_1, \pi_2\}$. That is op is any non-addressing function[2].

C.3 A partial ordering on address functions

We define a partial order, $<_a$, on the elements of *Addr*. $p <_a q$ for some $p, q \in Addr$ iff:

$$\begin{aligned}&(p = \pi_2) \wedge (q = f \cdot \pi_1)\vee\\&(p = f \cdot \pi_2) \wedge (q = g \cdot \pi_1)\vee\\&(p = f \cdot \pi_1) \wedge (q = g \cdot \pi_1) \wedge (f <_a g)\vee\\&q = \mathsf{K}\end{aligned}$$

informally, $<_a$ defines the partial ordering:

$$\begin{matrix}\pi_2\\ op \cdot \pi_2\end{matrix} <_a \begin{matrix}\pi_2 \cdot \pi_1\\ f \cdot \pi_2 \cdot \pi_1\end{matrix} <_a \begin{matrix}\pi_2 \cdot \pi_1 \cdot \pi_1\\ op \cdot \pi_2 \cdot \pi_1 \cdot \pi_1\end{matrix} <_a \ldots <_a \mathsf{K}$$

Note also that $\mathsf{K} <_a \mathsf{K}$ [3] and $\perp$ is not related to anything by $<_a$. Now we define a mapping, in two stages, from the index generating functions a_i appearing in nested index functions, to members of *Addr*.

[2]In non-translator code the set of addressing functions is bigger than the set $\{\pi_1, \pi_2\}$. This set would have to be enlarged to adapt these definitions to non-translator code.

[3]This means that the relative order of constant index-generating functions can be adjusted, with compensating use of transpose functions, with impunity. However, because K is the greatest element in the partial ordering, all constant index generating functions must be more deeply nested than any non-constant index-generating function in order to be part of a correctly formatted nested index function.

C.4 Extracting address functions from index-generators

First we define a function *ExtractAddrs* that maps each index generating function a_i to the set of address functions it contains:

$$ExtractAddrs(a_i) = \{p \in B\_EXP | addr\_seq(p, a_i) \wedge depends(a_i, p)\}$$

where the relation $address\_seq(p, a)$ holds iff:

$$\begin{array}{l}(a = (q_1, q_2)^\circ) \wedge (addr\_seq(p, q_1) \vee addr\_seq(p, q_2)) \vee \\ (p = g \cdot \pi_i) \wedge (a = f \cdot \pi_i) \wedge (addr\_seq(g, f)) \vee \\ (p = q = \pi_i) \vee \\ (a = op \cdot q) \wedge (Upstream(op) \neq \pi_i) \wedge addr\_seq(p, q)\end{array}$$

Where *Upstream* extracts the most upstream function in a composition sequence. $addr\_seq(p, q)$ is true if p matches a complete unbroken composed sequence of address functions in q. Under the earlier assumption that code is compacted, these unbroken sequences are the absolute addresses of values in the input tuple to the nested index function.

The second relation in *ExtractAddrs*: $depends(a, p)$ holds if:

$$\exists v \in VALUE, (a[p/Wrong]\, v = \perp) \wedge (a\, v \neq \perp)$$

where *Wrong* is a polymorphic function defined $Wrong\ x = \perp$. $depends(f)$ is true if f is ever evaluated for valid input values to the function[4]. If the address function is never evaluated then it is not added to the result of *ExtractAddrs*.

C.4.1 Condensing the output of ExtractAddrs

The second stage of mapping, is a function called *ToAddr*, that converts the output set of *ExtractAddrs* into either a single member of *Addr* or the value $\perp$. *ToAddr* is

[4] *depends* detects dead code. Such detection is generally undecidable. We are assuming that an implementation of *depends* will perform some conservative approximation of the semantics above where we assume dependency unless it can be proven dependency does not exist. Also note, that if a stage of dead-code-elimination is implemented prior to vector-optimisation the call to *depends* can be removed.

defined:

$$\begin{array}{lcl} \mathit{ToAddr}(\emptyset) & = & \mathsf{K} \\ \mathit{ToAddr}(\{p\}) & = & p, \textbf{\textit{if}}\, p \in \mathit{Addr} \\ \mathit{ToAddr}(\{f\}) & = & \bot, \textbf{\textit{if}}\, f \notin \mathit{Addr} \\ \mathit{ToAddr}(A) & = & \bot, \textbf{\textit{if}}\, \mathit{Cardinality}(A) > 1 \end{array}$$

The first clause returns the place-holder K if the set of extracted functions is empty. The second clause returns the address function contained in the set. The third clause handles invalid address functions, this clause should not apply to valid, compacted, translator code. The fourth clause applies when two or more different address functions are extracted.

C.5 The property of being sorted

Given the definitions above, it is now possible to define what makes a nested indexing function sorted. The nested indexing function:

$$! \cdot (! \cdot (\ldots (! \cdot (v, a_{n-1})^\circ, \ldots)^\circ, a_1)^\circ, a_0)^\circ$$

is sorted iff:

$$\begin{array}{l} \mathit{ToAddr}(\mathit{ExtractAddrs}(a_0)) <_a \mathit{ToAddr}(\mathit{ExtractAddrs}(a_1) <_a \ldots \\ \quad <_a \mathit{ToAddr}(\mathit{ExtractAddrs}(a_{n-1})) \end{array}$$

That is, if the outermost index generator is least in the partial order and the next outermost is second-least in so on. Intuitively, it means that π_2 functions must be on the outside and more complex addressing functions should be further inside and index-generators not dependent on any input value should be innermost.

C.6 From sortable to sorted

The section on reorientation on page 127 showed that it is possible, in some cases, to insert tranpose functions into code in order to make the elimination of index operations possible. In other words, transpose can be used to turn an unsorted indexing function into a sorted indexing function. Indexing functions that can be converted in this way are *sortable*.

An indexing function:

$$! \cdot (! \cdot (\ldots (! \cdot (v, a_{n-1})^{\circ}, \ldots)^{\circ}, a_1)^{\circ}, a_0)^{\circ}$$

is sortable iff there is some permutation:

$$p = \mathit{permute}[a_0, a_1, \ldots, a_{n-1}]$$

of the index-generators for which:

$$\begin{aligned}&\mathit{ToAddr}(\mathit{ExtractAddrs}(p_0)) <_a \mathit{ToAddr}(\mathit{ExtractAddrs}(p_1) <_a \cdots\\ &\quad <_a \mathit{ToAddr}(\mathit{ExtractAddrs}(p_{n-1}))\end{aligned}$$

That is, a nested indexing function is sortable if the partial ordering $>_a$ can be applied to some permutation of its extracted and condensed addresses.

If it exists, a sorted permutation of index generators is achieved by using applications of transpose to swap the order of index generators.

A final note All of the preceding definitions depend on the tuple structure built by translator code. The definitions will change with major changes to the structure of code produced by the translator. In most cases, minor changes to the language such as the addition of new primitives will not affect the definitions above.

Appendix D

The time-space model

This appendix describes a sequential model for the execution of BMF programs against known data. The model is defined as a series of rules. Each rule is associated with a BMF function. When combined, the rules form a specification for an interpreter. This interpreter not only produces an output value but other information about the program's execution. It is, essentially, a *collecting interpretation*[82] over BMF programs and their input data.

Amongst the other data collected by the interpreter is a trace of the the space consumption of the program over time. This trace information is used in chapters 4 and 5 as a basis for a comparison of the efficiency of BMF programs. An executable version of the interpreter has been implemented in Miranda^{tm}, and this version corresponds to the rules shown in this appendix.

The layout of this appendix is as follows. Section D.1 defines the syntactic domains used to describe the interpreter and its input. Section D.2 describes the semantics of the top level rules of the interpreter and some auxiliary rules. Section D.3 presents the rules of the interpreter and forms the bulk of this appendix. Lastly, section D.4 very briefly describes the process of extracting a trace from the final state produced from the interpreter.

D.1 Syntax

The syntax used by the interpreter is split between three syntactic domains (phyla). The first domain is the *VALUE* (see chapter 4). *VALUE* contains the syntax of valid input values for BMF functions. *VALUE* contains syntax for integer, real and boolean values as well as for aggregate structures i.e. tuples and vectors. For the purposes of this interpreter, we define an additional member of the *VALUE* domain called *null*. *null* is a place-holder for a not-yet-defined value.

The second domain is *B_EXP* (see chapter 4). *B_EXP* defines the syntax of BMF functions.

The third domain contains the syntax for defining interpreter rules. This domain is called *I_RULES*. It contains the sequence of interpreter rules defining how the trace is generated for each BMF function. The first rule in this sequence belongs to the *I_TOP* phyla. Other rules, at least one for each BMF function, are contained in the *I_RULE* phyla. An (almost complete) abstract syntax for interpreter rule definitions is given below.

Phyla

I_RULES $\mapsto$ *Interpreter Rules*, *I_TOP* $\mapsto$ *Top rule*,
I_RULE $\mapsto$ *Interpreter Rule*, *I_DEF* $\mapsto$ *Interpreter definition*,
I_EXP $\mapsto$ *Interpreter expressions*, *I_PAT* $\mapsto$ *Patterns*,
I_STATE $\mapsto$ *Interpreter state*, *I_TRACE* $\mapsto$ *Interpreter trace*,
I_TR $\mapsto$ *Trace record*, *I_NUM* $\mapsto$ *Number ref*,
I_NAME $\mapsto$ *Name*, *NUM* $\mapsto$ *Number*,
VALUE $\mapsto$ *Value*, *B_EXP* $\mapsto$ *BMF functions*.

Abstract syntax

I_RULES	::= `irules` *I_TOP* $[I\_RULE]^+$
I_TOP	::= `tracegen` *I_PAT* *I_EXP*
I_RULE	::= `strdef` *I_PAT* *I_EXP*
I_DEF	::= `def` *I_NAME* $[I\_PAT]^*$ *I_EXP*.
I_EXP	::= `pass_between` $[I\_NAME]^+$ *I_STATE* | `space` *I_PAT* | `add_to_trace` *I_STATE* *I_NUM* *I_NUM* | `length` *I_PAT* | `append` *I_TRACE* *I_TRACE* | `pval` *I_PAT* | `plus` *I_NUM* *I_NUM* | `mult` *I_NUM* *I_NUM* | `minus` *I_NUM* *I_NUM* | `uminus` *I_NUM* | `funapp` *I_NAME* $[I\_PAT]^+$ | `where` *I_EXP* $[I\_DEF]^+$ | `ival` *I_PAT* | `str` *I_PAT*.

I_PAT	::= `var` *I_NAME* | `val` *VALUE* | `ilist` [*I_PAT*]* | `ituple` [*I_PAT*]$^{+2}$ | `bmf` *B_EXP*.
I_STATE	::= `state` *I_NUM I_NUM I_PAT I_PAT I_TRACE*.
I_TRACE	::= `trace` [*I_TR*]* | `tr_name` *I_NAME*.
I_TR	::= `tr` *I_NUM I_NUM*.
I_NUM	::= `inum` *NUM* | `numvar` *I_NAME*.
I_NAME	::= Implemented as a *string*.
NUM	::= Implemented as an *integer*.
VALUE	::= see definition from chapter 4 + `null`.
B_EXP	::= see definition from chapter 4.

Notes on syntax Each interpreter rule (*I_RULE*) must have an *I_PAT* term of the form `ituple`[*B_EXP*,i_pat_0] where i_pat_0 must be an *I_PAT* of the form `val` *VALUE* or `var` *I_NAME*.

The main *I_PAT* term belonging to each interpreter rule (*I_RULE*) always has the form `ituple`[*B_EXP*,i_pat_0] where i_pat_0 must be an *I_PAT* of the form `val` *VALUE* or `var` *I_NAME*.

Concrete syntax The mapping between expressions in the abstract syntax and their concrete equivalents is shown below. The rules of the interpreter will be defined using the concrete forms.

Abstract Syntax		**Concrete Syntax**
`irules` *trg* $[r_0, ..., r_{n-1}]$	$\mapsto$	*trg* $r_0, \ldots, r_{n-1}$
`tracegen` (b_exp, v)	$\mapsto$	*Trg*(b_exp,v)
`strdef` (b_exp, st) *exp*	$\mapsto$	$Str(b\_exp, st) = exp$
`def` a $[p_0, ..., p_{n-1}]$ *exp*	$\mapsto$	$a\ p_0 \ldots p_{n-1} = exp$
`pass_between` $[f_0, ..., f_{n-1}]$ *env*	$\mapsto$	*pass_between* $[f_0, ..., f_{n-1}]$ *env*
`space` *pat*	$\mapsto$	$S(pat)$
`length` *pat*	$\mapsto$	$\#pat$
`add_to_trace` *env* n_0 n_1	$\mapsto$	*add_to_trace env* n_0 n_1
`append` tr_0 tr_1	$\mapsto$	$tr_0 {+\!+}\, tr_1$
`pval` *st*	$\mapsto$	$Pval(st)$
`plus` n_0 n_1	$\mapsto$	$n_0 + n_1$

`mult` n_0 n_1	$\mapsto$	$n_0 \times n_1$
`minus` n_0 n_1	$\mapsto$	$n_0 - n_1$
`uminus` n	$\mapsto$	$-n$
`funapp` a $[p_0, ..., p_{n-1}]$	$\mapsto$	$a\ p_0, \ldots, p_{n-1}$
`where` exp $[d_0, ..., d_{n-1}]$	$\mapsto$	exp ***where*** $d_0, \ldots, d_{n-1}$
`ival` p	$\mapsto$	p
`str` p	$\mapsto$	$Str(p)$
`var` a	$\mapsto$	a
`val` v	$\mapsto$	v
`ilist` $[v_0, ..., v_{n-1}]$	$\mapsto$	$[v_0, ..., v_{n-1}]$
`ituple` $[v_0, ..., v_{n-1}]$	$\mapsto$	$(v_0, ..., v_{n-1})$
`state` $s\ t\ i\ o\ tr$	$\mapsto$	(s, t, i, o, tr)
`trace` $[tr_0, ..., tr_{n-1}]$	$\mapsto$	$[tr_0, ..., tr_{n-1}]$
`tr_name` a	$\mapsto$	a
`tr` n_0 n_1	$\mapsto$	(n_0, n_1)
`inum` n	$\mapsto$	n
`numvar` a	$\mapsto$	a
`null`	$\mapsto$	$null$.

Other concrete syntax The concrete syntax corresponding to *VALUE* and *B_EXP* is shown in chapter 4.

D.2 Semantics

There will is no formal semantic description of the interpretation or trace building process in this report. Instead, the syntax defined above will be used as a vehicle to present the rules of the BMF interpreter[1].

In this section we define, informally, the meaning of some of the framework for the interpreter rules.

[1]To complete a formal description of the interpretation process an interpreter for interpreter rules could be defined in natural semantics. We have omitted such a definition for now. This decision is justified by the fact that the interpreter rules are defined in informal mathematical notation and, as such, can be understood without a lengthy introduction.

D.2.1 The trace generator

Trg is the function forming the entry-point of the time-space interpreter. *Trg* accepts a tuple containing a BMF function and a value, injects the value into an initial interpreter state and passes the resulting state and the BMF function into the set of trace generating rules which, eventually, generates a final interpreter state.

Type

$$(\textit{BMF function}, \textit{Value}) \rightarrow \textit{Interpreter state}$$

Semantics The semantics of *Trg* is defined by the following expression:

$$\textit{Trg}\ (b\_exp, v) = \textit{Str}(b\_exp, (S(v), 0, v, \textit{null}, [\,]))$$

That is, the result of *Trg* is the same as the result of *Str* with the appropriate initial values injected into the starting interpreter state. This state consists of an an initial space requirement $S(v)$, an initial time 0, an initial input value v, an initial output value *null*, and an initial empty trace $[\,]$. Note that the *null* value is used as a place-holder when no valid value yet exists[2].

D.2.2 Interpreter rules

Str is the state-transforming function. *Str* is defined by the rules of the interpreter. Each rule of the interpreter defines *Str* with respect to a different BMF construct.

Type

$$(\textit{BMF function}, \textit{Interpreter state}) \rightarrow \textit{Interpreter state}$$

where the interpreter state consists of the *current* space consumption, current time, input value, output value and trace. The trace is a series of integer pairs of the form: $(\textit{space}, \textit{time})$. Each point represents an instantaneous transition[3] to a new value for space consumption at the corresponding time. *Str* is the core of the interpreter.

Semantics The semantics of *Str* are defined in the interpreter rules in the next section.

[2]*null* is similar to $\perp$.

[3]Hence the use of discrete steps to graph traces in chapters 4 and 5.

D.2.3 Auxiliary function definitions

Auxiliary functions are called by *Str* to assist in the process of transforming the state. Definitions of these functions follow

D.2.3.1 *pass_between*-passing the state

pass_between is a function used to pass an state tuple between a series of functions. *pass_between* is used where the execution of a BMF construct can be expressed as a composition of state transforming functions.

Type

$$([(IS \to IS)], IS) \to IS$$

where *IS* is an abbreviation for *Interpreter state*.

Semantics If we view the first input parameter is a cons-list a recursive definition of *pass_between* is:

$$\begin{aligned} pass\_between\ [\,]\ st &= st \\ pass\_between\ (f : fs)\ st &= pass\_between\ fs\ (f\ \ st) \end{aligned}$$

D.2.3.2 *S*-the space function

S is a function used to determine the space consumed by a value.

Type

$$Value \to Number$$

Semantics $S(v)$ will return an integer value being the space, in words[4], currently occupied by v. *S* is defined by a small set of rules corresponding to the recursive structure of *VALUE*.

D.2.3.3 *add_to_trace*-adding new trace records

add_to_trace creates an updated state with new space, time and trace values.

[4]A *word* is defined here as the space consumed by a single integer or floating point number. For the sake of convenience, all scalar values are assumed to consume one word of space.

Type

$$(\mathit{Interpreter\ state}, \mathit{Number}, \mathit{Number}) \rightarrow \mathit{Interpreter\ state}$$

Semantics A definition of *add_to_trace* is:

$$\begin{aligned} \mathit{add\_to\_trace}\ (s, t, i, o, tr)\ si\ ti = \ & (s + si, t + ti, i, o, tr +\!\!+ [(sl + si, tl + ti)]) \\ & \mathit{where} \\ & sl = (\mathit{fst} \cdot \mathit{last})\ st \\ & tl = (\mathit{snd} \cdot \mathit{last})\ st \end{aligned}$$

where *last* accesses the last element in a trace list, *fst* accesses the first element of a pair and *snd* accesses the second element of a pair. Note that *si* and *ti* are increments on the current (last) trace values rather than being absolute space and time values.

D.2.3.4 *add_to_trace_null*-adding new trace records and setting input to *null*

This function is identical to *add_to_trace* except that in addition to updating the time and space values it sets the input value to null. It is defined:

$$\begin{aligned} \mathit{add\_to\_trace\_null}\ (s, t, i, o, tr)\ si\ ti = \ & (s + si, t + ti, \mathit{null}, o, tr +\!\!+ [(sl + si, tl + ti)]) \\ & \mathit{where} \\ & sl = (\mathit{fst} \cdot \mathit{last})\ st \\ & tl = (\mathit{snd} \cdot \mathit{last})\ st \end{aligned}$$

add_to_trace_null is used at the end of rules where, by convention, the input value is set to null to record its deallocation.

D.2.3.5 *append*-combining two traces

append, which is written as concatenate ($+\!\!+$) joins two trace lists together. *append* does not check that the time in the first trace record of the second list is greater than or equal to the time in the last trace record of the first list.

Type

$$(\mathit{Interpreter\ Trace}, \mathit{Interpreter\ Trace}) \rightarrow \mathit{Interpreter\ Trace}$$

Semantics The semantics of *append* corresponds exactly to that of list concatenate. Informally:

$$[x_0, ..., x_i] +\!\!+ [y_0, ..., y_j] = [x_0, ..., x_i, y_0, ..., y_j]$$

D.2.3.6 Length Operator

The length operator (#) is used to find the length of an input vector.

Type

$$\mathit{Value} \rightarrow \mathit{Num}$$

Semantics Assuming the input is a vector the definition of # is:

$$\begin{aligned} \#\,[\,] &= 0 \\ \#\,[x] &= 1 \\ \#xs +\!\!+\, ys &= (\#xs) + (\#ys) \end{aligned}$$

D.2.3.7 Projecting the output value

The operator *Pval* projects the input value component out of the interpreter state.

Type

$$\mathit{I\_STATE} \rightarrow \mathit{VALUE}$$

Semantics *Pval* extracts the fourth element of the interpreter state tuple. It is defined:

$$\mathit{Pval}(s, t, i, o, tr) = o$$

D.2.3.8 Arithmetic Operators

The operators +, × and minus (−) have their conventional meanings. They are used for arithmetic over time and space values.

Semantics The conventional semantic definitions for these operators apply. Variable references are resolved to *Num* before these operators are applied.

D.3 Interpreter Rules

The rules of the interpreter form the core of this chapter. They describe the process for generating the sequential traces used in this work. The rules will be presented shortly but, in order to motivate their structure, it is worth reiterating the conventions that guide their construction.

Conventions used by the interpreter The following conventions apply to the sequential performance model:

- Storage for output of each scalar function is allocated just prior to evaluation of that function.
- Storage for input data of each scalar function is de-allocated just after evaluation of each function.
- The time taken to allocate or deallocate any amount of storage is one time unit.
- With the above in mind, functions on scalar data, such as $+$ and $\times$ have the following execution profile:
    1. allocate storage for the result (one time unit).
    2. perform the operation (one time unit).
    3. deallocate storage for the input (one time unit).
- A vector takes up one unit of space plus the combined space used by its elements.
- A tuple takes up a space equal to the combined space used by its elements.
- The cost of copying one item of scalar data is one time unit.
- During aggregate operations, data is copied/allocated at the last possible moment and deallocated at the first possible moment.

These conventions guide the design of the rules that follow. There is a rule for each BMF construct. A short introduction to the semantics of each construct is given before its rule. Each rule is followed by a synopsis of its main features.

D.3.1 Scalar Functions

Scalar functions include arithmetic, logical, trigonometric and type-coercion functions. All of these functions produce an output value which is one word long. These functions have rules with the following form for any given scalar function f:

$$
\begin{array}{l}
\mathit{Str}(f,\quad (s,t,v,o,tr)) = \mathit{last\_part} \\
\qquad \textbf{\textit{where}} \\
\qquad \begin{array}{lcl}
\mathit{first\_part} & = & \mathit{add\_to\_trace}\,(s,t,v,o,tr)\;1\;1 \\
\mathit{last\_part} & = & \mathit{applyf}\;\mathit{first\_part} \\
\mathit{applyf}\,(s,t,v,o,tr) & = & (s1, t1, \mathit{null}, f\;v, tr +\!\!+ [(s1,t1)])
\end{array} \\
\qquad\qquad \textbf{\textit{where}} \\
\qquad\qquad \begin{array}{lcl}
s1 & = & s - S(v) \\
t1 & = & t + 2
\end{array}
\end{array}
$$

first_part is the state after allocating the word of space required for the result. *last_part* is the state after applying f to its input value v and then deallocating the input value. *last_part* is the state returned by the scalar function.

D.3.2 Length

The length function takes a vector as input and produces an integer representing its length. Even though its input is not a scalar, the rule for length is the same as the one for scalar functions above with length substituted for f.

D.3.3 Vector Indexing

The index function takes pair (vs, i) as input, where vs is a vector and i is an integer and returns the value vs_i as a result. The rule for vector indexing is

$$\begin{aligned}
&Str(\mathsf{index}, \quad (s,t,(vs,i),o,tr)) = \mathit{last\_part}\\
&\qquad \textbf{\textit{where}}\\
&\qquad \mathit{first\_part} &&= \mathit{add\_to\_trace}\ (s,t,(vs,i),o,tr)\ svsi\ 1\\
&\qquad \mathit{last\_part} &&= \mathit{apply\_index}\ \mathit{first\_part}\\
&\qquad svsi &&= S(vs!i)\\
&\qquad \mathit{apply\_index}(s,t,(vs,i),o,tr) &&= (s1,t1,\mathit{null},vs!i,tr{+}{+}\ [(s1,t1)])\\
&\qquad\qquad \textbf{\textit{where}}\\
&\qquad\qquad s1 = s-(S(vs)+S(i))\\
&\qquad\qquad t1 = t+svsi
\end{aligned}$$

The state *first_part* accounts for the allocation of space to hold the result of the index function. The state *last_part* accounts for the cost of copying the indexed value across to the space allocated for the result and for the deallocation of the original input vector and index value.

D.3.4 The identity function

The id function returns its input value. If we strictly adhere to the conventions followed in the rest of the interpreter then the rule for id is:

$$\begin{aligned}
&Str(\mathsf{id}, \quad (s,t,v,o,tr)) = \mathit{last\_part}\\
&\qquad \textbf{\textit{where}}\\
&\qquad \mathit{first\_part} &&= \mathit{add\_to\_trace}\ (s,t,v,o,tr)\ S(v)\ 1\\
&\qquad \mathit{last\_part} &&= \mathit{apply\_id}\ \mathit{first\_part}\\
&\qquad \mathit{apply\_id}\ (s,t,v,o,tr) &&= (s-S(v),t+S(v),\mathit{null},v,tr{+}{+}\ [(s-S(v),t+S(v))])
\end{aligned}$$

However, if we are willing to make a sensible departure from convention in this case then the rule could be simplified to:

$$Str(\mathsf{id},\ (s,t,v,o,tr)) = (s,t+1,\mathit{null},v,tr{+}{+}\ [(s,t+1)])$$

which just increments the time consumption by one unit. This, simpler version, is used in the model used to generate the traces in this report[5].

[5]Our motivation for departing from convention is that strict adherence to the convention, by using the first version of the rule, would result in sometimes quite large costs being attached to id. These costs are easily avoided even in a relatively naive implementation. These unrealistic costs would impact disproportionately on translator code, due to the number of id functions that such code typically contains. In the interests of a fair comparison we do not use the expensive rule for id in our experiments.

D.3.5 Function Composition

The b_comp(f,g) function, written $f \cdot g$ takes an input value v and produces the result $f(g\,v)$.

$$\begin{array}{l} Str(f \cdot g, \quad st) = result \\ \quad \textbf{\textit{where}} \\ \quad \begin{array}{lcl} (s,t,i,o,tr1) & = & Str(f\ st) \\ result & = & Str(g\ (s,t,o,null,tr1)) \end{array} \end{array}$$

Synopsis The first line under the ***where*** clause applies the function g to the input state st to produce a new state $(s,t,i,o,tr1)$. The next line takes this state, with the output value, o, moved to the input position, applies it to the function f producing the state *result* which is the final result of this rule.

D.3.6 Map

The map(g) function takes an input vector $[x_0,\ldots,x_{n-1}]$ an applies g point-wise to produce a new vector as a result.

$$\begin{array}{l} Str(\mathsf{map}\quad g,\ (s,t,[x_0,...,x_{n-1}],o,tr)) = last\_part \\ \quad \textbf{\textit{where}} \\ \quad \begin{array}{lcl} first\_part & = & add\_to\_trace\,(s,t,[x_0,...,x_{n-1}],[\,],tr)\,1\,1 \\ main\_part & = & pass\_between[f_0,\ldots,f_{n-1}]\,first\_part \\ last\_part & = & add\_to\_trace\_null\ main\_part\ -1\,1 \end{array} \\ \quad f_i\,(s,\quad t,[x_i,...,x_{n-1}],[v_0,...,v_{i-1}],tr) \\ \qquad = \ (s1,t1,[x_{i+1},...,x_{n-1}],[v_0,...,v_i],tr1) \\ \qquad\quad \textbf{\textit{where}} \\ \qquad\quad (s1,t1,null,v_i,tr1) \ = \ Str(g,(s,t,x_i,null,tr)) \end{array}$$

Synopsis *first_part* is generated by a call a call to *add_to_trace*. This call increments the initial time and space measures by one to account for the allocation of an empty result vector. The result vector will grow incrementally as the rule for map produces new results.

main_part simulates the application of g point-wise to the input vector. The call to *pass_between* transfers intermediate states between the list of auxiliary functions $[f_0,\ldots,f_{n-1}]$. Each of these functions applies g to the next element of the input vector and produces a new element of the output vector. Each input element is deallocated as soon as it is used. Space for each output element is allocated just before it is produced. Note that the rule does not define the $[f_0, \ldots, f_{n-1}]$ functions individually. Instead a single function, parameterised by i, called f_i is defined to capture their meaning.

Finally, the *last_part* state accounts for the deallocation of the now-empty input vector after all of the point-wise applications of g have taken place.

D.3.7 Distl

The distl function takes a pair $(v, [x_0, \ldots, x_{n-1}])$ and produces a vector $[(v, x_0), \ldots, (v, x_{n-1})]$ consisting of copies of the first element of the pair distributed over the second element of the pair.

$$
\begin{array}{l}
Str(\mathsf{distl}\ ,(s, t, (v, [x_0, ..., x_{n-1}]), o, tr)) = last\_part \\
\quad \textbf{\textit{where}} \\
\quad \begin{array}{lcl}
first\_part & = & add\_to\_trace\ (s, t, (v, [x_0, ..., x_{n-1}]), [\,], tr)\ 1\ 1 \\
main\_part & = & pass\_between[f_0, ..., f_{n-1}]\ first\_part \\
last\_part & = & add\_to\_trace\_null\ main\_part\ -(sv + 1)\ 1 \\
sv & = & S(v)
\end{array} \\
\quad f_i\ \ (s, t, (v, [x_i, ..., x_{n-1}]), [(v, x_0), ..., (v, x_{i-1})], tr) = \\
\qquad (s1, t1, (v, [x_{i+1}, ..., x_{n-1}]), [(v, x_0), ..., (v, x_i)], tr +\!\!+ [(s1, t1)]) \\
\qquad \textbf{\textit{where}} \\
\qquad \begin{array}{lcl}
s1 & = & s + sv \\
t1 & = & t + sv + S(x_i)
\end{array}
\end{array}
$$

Synopsis The state *first_part* accounts for the allocation of the new output vectors. *main_part* simulates the production of the output vector from copies of v and each x_i. Each x_i is deallocated as soon as its value is transferred to the result vector. v is not deallocated because its value is needed again.

Finally, *last_part*, accounts for the deallocation of v once the production of the entire output vector is complete.

D.3.8 Reduce

The reduce($\oplus, z$) function takes an input vector $[x_0, \ldots, x_{n-1}]$ and produces the output value $x_0 \oplus \ldots \oplus x_{n-1}$. If the input vector is empty the value produced is z applied to $[x_0, \ldots, x_{n-1}]$.

```
Str(reduce(⊕, z),  (s, t, [], o, tr) = zero_result
                   where
                   zero_result = Str(z, (s, t, [], o, tr))
Str(reduce(⊕, z),  (s, t, [x_0, ..., x_{n-1}], o, tr)) = last_part
                   where
                    main_part  =  pass_between [f_1, ..., f_{n-1}] (s, t, [x_1, ..., x_{n-1}], x_0, tr)
                    last_part  =  add_to_trace_null main_part  −1 1
                    f_i(s,  t, [x_i, ..., x_{n-1}], v, tr) =
                           (s1, t1, [x_{i+1}, ..., x_{n-1}], v1, tr1)
                           where
                           (s1, t1, null, v1, tr1) = Str(⊕, (s, t, (v, x_i), null, tr))
```

Synopsis The interpretation of reduce can take one of two different paths. If the input value is an empty vector, then the first rule applies the zero-element z to the input state[6]). to produce the state labelled *zero_result*.

If the input value is a non-empty vector then the second rule is used. *main_part* simulates the application of $\oplus$ to successive elements of the input vector. Each intermediate result is used as the left-operand of the next application. Values of the input vector are allocated as they are consumed. The *last_part* state accounts for the deallocation of the empty input vector at the end of processing.

The interpreter has similar rules for reducel. In the rule for reducer the last element of the vector is operated upon first and the tuple (v, x_i) is reversed. For reducep, reducelp and reducerp the first clause of the rule for handling empty input vectors is omitted.

[6]The translation process ensures that the zero-element, when it remains with the reduce function, is a constant function (see page 74.

D.3.9 Scan

The $\mathsf{scan}(\oplus)$ function takes an input vector $[x_0, ..., x_{n-1}]$ and produces an output vector $[x_0, x_0 \oplus x_1, \ldots, x_0 \oplus x_1 \oplus \ldots \oplus x_{n-1}]$ of the cumulative application of $\oplus$ to the prefixes of the input vector.

$$
\begin{array}{l}
\mathit{Str}(\mathsf{scan}\ (\oplus),\ (s, t, [\,], o, tr)) = \mathit{empty\_result} \\
\qquad \textbf{\textit{where}} \\
\qquad \mathit{empty\_result} = \mathit{add\_to\_trace\_null}\ (s, t, \mathit{null}, [\,], tr)\ 0\ 2 \\
\mathit{Str}(\mathsf{scan}\ (\oplus),\ (s, t, [x_0, ..., x_{n-1}], o, tr)) = \mathit{last\_part} \\
\qquad \textbf{\textit{where}} \\
\qquad \begin{array}{lcl}
\mathit{first\_part} & = & \mathit{add\_to\_trace}\ (s, t, [x_1, ..., x_{n-1}], [x_0], tr)\ 1\ 1 + S(x_0) \\
\mathit{main\_part} & = & \mathit{pass\_between}\ [f_1, ..., f_{n-1}] \\
 & & \qquad (s, t, [x_0, ..., x_{n-1}], [x_0], tr) \\
\mathit{last\_part} & = & \mathit{add\_to\_trace\_null}\ \mathit{main\_part}\ -1\ 1
\end{array} \\
\qquad f_i(s, t,\ [x_i, ..., x_{n-1}], [v_0, ..., v_{i-1}], tr) = \\
\qquad\qquad (s1, t1, [x_{i+1}, ..., x_{n-1}], [v_0, ..., v_{i-1}, v_i], tr1) \\
\qquad\qquad \textbf{\textit{where}} \\
\qquad\qquad (s1, t1, di, v_i, tr1) = \mathit{Str}(\oplus, (s, t, (v_{i-1}, x_i), \mathit{null}, tr))
\end{array}
$$

Synopsis If the input vector is empty the first clause of the rule for scan will allocate a new empty vector for a result and deallocate the empty input vector[7] returning the new state *empty_result*.

If the input vector is not empty then the second clause applies. The *first_part* state accounts for the copying of the first element of the input vector to the new result vector. Note that net space consumption is increased by one to account for the allocation of a new vector descriptor and time is increased to account for this allocation plus the copying of the first element.

main_part simulates the application of $\oplus$ to successive intermediate results and successive elements of the input vector. For each application of $\oplus$ the current intermediate result is applied to the next input value. The result becomes a new intermediate value which is copied to the output vector. The old input value is then deallocated.

[7] In this case, allocation and deallocation is not necessary. It is done in this rule for consistency with the conventions used in other parts of the interpreter.

Finally, after *main_part* has been produced *last_part* accounts for the deallocation of the input vector.

The definitions for scanl, scanp and scanlp, are similar to the rules above. The definitions for scanr and scanrp are more verbose but their infrequent use does not warrant describing them here.

D.3.10 Alltup functions

An alltup function $(g_1, \ldots, g_n)^\circ$ applies a copy of its input value v to each g_i producing a new tuple, $(g_1\, v, \ldots g_n\, v)$.

```
Str( (g_1,...,g_n)°, (s,t,v,o,tr)) = last_part
    where
      main_part  =  pass_between [f_1,...,f_{n-1}] (s,t,v,o,tr)
      last_part  =  applye main_part
      f_i (s, t,v,(o_1,...,o_{i-1}),tr) = (s2,t2,v,(o_1,...o_i),tr2)
          where
            (s2,t2,v,o_i,tr2)  =  Str(g_i, (s1,t1,v,null,tr1))
            sv                 =  S(v)
            tr1                =  add_to_trace (s1,t1,v,(o_1,...,o_{i-1}),tr) sv sv
      applye (s,t,v,(o_1,...,o_{n-1}),tr) = (s1,t1,null,(o_1,...,o_n),tre)
            where
            (s1,t1,null,o_n,tre) = Str(g_n, (s,t,v,null,tr))
```

Synopsis *main_part* simulates the application of each g_i, except the last, to a copy of the input value v. A new copy of v is made immediately prior to each of these applications and this copy is deallocated by that application.

The final state is generated by a call to the local function *applye*. *applye* simulates the application of the last function in the alltup, g_n to the original input value. This remaining input value is deallocated by this last application producing the final state for this rule.

D.3.11 Allvec functions

An allvec function $[f_0, \ldots, f_{n-1}]^\circ$ applies its input value v to each f_i producing a vector, $[f_0\, v, \ldots, f_{n-1}\, v]$. Note that all f_i must have the same type[8]. If the allvec function is empty, that is if it takes the form $[\,]^\circ$, then it will return an empty vector irrespective of its input.

$$
\begin{array}{l}
Str([\quad]^\circ,\ (s,t,v,o,tr)) = (s,t,v,[\,],tr2) \\
\qquad \textbf{\textit{where}} \\
\qquad \begin{array}{lcl}
tr1 & = & (s,t,null,[\,],tr) \\
tr2 & = & add\_to\_trace\ tr1\ (-sv+1)\ 2 \\
sv & = & S(v)
\end{array} \\
Str([\ f_0, ..., f_{n-1}]^\circ,\ st) = last\_part \\
\qquad \textbf{\textit{where}} \\
\qquad \begin{array}{lcl}
main\_part & = & pass\_between\ [f_0, ..., f_{n-2}]\ st \\
last\_part & = & applye\ main\_part
\end{array} \\
\qquad f_i\,(s,\ \ t,v,[o_0, ..., o_{i-1}],tr) = (s2,t2,v,[o_0, ...o_i],tr2) \\
\qquad\qquad \textbf{\textit{where}} \\
\qquad\qquad \begin{array}{lcl}
(s2,t2,v2,o_i,tr2) & = & Str(f_i,\ (s1,t1,v,null,tr1)) \\
sv & = & S(v) \\
tr1 & = & add\_to\_trace\ (s1,t1,v,[o_0, ..., o_{i-1}],tr)\ sv\ sv
\end{array} \\
\qquad applye\ \ (s,t,v,[o_0, ..., o_{n-2}],tr) = (s1,t1,null,[o_0, ..., o_{n-1}],tr1) \\
\qquad\qquad \textbf{\textit{where}} \\
\qquad\qquad (s1,t1,null,o_{n-1},tr1) = Str(f_{n-1},\ (s,t,v,null,tr))
\end{array}
$$

Synopsis The interpreter handles allvec expressions in a very similar manner to the way it handles alltup functions. The primary difference between the allvec and alltup rules is the clause to handle an empty allvec function. This clause updates the trace to reflect the deallocation of the input value and the allocation of space for the empty result vector.

The handling of non-empty allvec functions is structurally identical to the handling of alltup functions.

[8]This restriction is not enforced by the interpreter, it is assumed to be enforced by the typechecker of the Adl compiler.

D.3.12 Select

select takes a pair of vectors $(xs, [y_0, \ldots, y_{n-1}])$ and produces a new vector $[xs!y_0, \ldots, xs!y_{n-1}]$. That is, the result produced comprises the elements of the first vector indexed by the second vector.

$$\begin{array}{l}
Str(\ \text{select}, (s, t, (xs, [y_0, ..., y_{n-1}]), o, tr)) = last\_part \\
\quad \textbf{\textit{where}} \\
\quad \begin{array}{lcl}
first\_part & = & add\_to\_trace\ (s, t, (xs, [y_0, ..., y_{n-1}]), null, tr)\ 1\ 1 \\
main\_part & = & pass\_between\ [f_0, ..., f_{n-1}]\ (s, t, (xs, [y_0, ..., y_{n-1}]), null, tr) \\
last\_part & = & add\_to\_trace\_null\ main\_part\ -(S(xs) + 1)\ 2
\end{array} \\
\quad f_i\,(s, t,\ \ (xs, [y_i, ..., y_{n-1}]), [v_0, ..., v_{i-1}], tr) = \\
\qquad\qquad (s1, t1, (xs, [y_{i+1}, ..., y_{n-1}]), [v_0, ..., v_i], tr1) \\
\qquad\qquad \textbf{\textit{where}} \\
\qquad\qquad \begin{array}{lcl}
v_i & = & xs!y_i \\
s1 & = & s + S(v_i) - S(y_i) \\
t1 & = & t + S(v_i) + 1 \\
tr1 & = & tr +\!\!+ (s1, t1)
\end{array}
\end{array}$$

Synopsis *first_part* increments the time and space from the initial state to account for the allocation of an empty result vector. *main_part* simulates the copying of vector elements from xs into the result vector, deallocating each y_i as it goes. *last_part* decrements the space consumption to account for the deallocation of xs and the, now-empty, y-vector.

D.3.13 Zip

The zip function takes a pair of vectors of equal length $([x_0, \ldots, x_{n-1}], [y_0, \ldots, y_{n-1}])$ and produces a vector of pairs: $[(x_0, y_0), \ldots, (x_{n-1}, y_{n-1})]$.

```
$St(\mathsf{zip},\ \ (s, t, (xs, ys), null, tr)) = last\_part$
    $\textbf{where}$
    $n \quad = \#xs$
    $first\_part \quad = add\_to\_trace\,(s, t, (xs, ys), null, tr)\ 1\ 1$
    $last\_part \quad = add\_to\_trace\_null\ main\_part\ -2\ 2$
    $main\_part \quad = pass\_between\,[f_0, ..., f_{n-1}]\,first\_part$
    $f_i\ \ (s, t, ([x_i, ..., x_{n-1}], [y_i, ..., y_{n-1}]), [(x_0, y_0), ..., (x_{i-1}, y_{i-1})], tr) =$
        $(s, t1, ([x_{i+1}, ..., x_{n-1}], [y_{i+1}, ..., y_{n-1}]), [(x_0, y_0), ..., (x_i, y_i)], tr1)$
        $\textbf{where}$
        $tr1 \quad = add\_to\_trace\ tr\ 0\ t1$
        $t1 \quad = t + (S(x_0) + S(y_0)) + 1$
```

Synopsis *first_part* simulates the allocation of the new result vector. *main_part* simulates the copying of corresponding x_i and y_i values into pairs in the result vector. Note that *tr1* increments space consumption by zero. This is due to allocation and deallocating cancelling each other out. Note that $t1$ takes account of the time consumed by copying x_i and y_i.

Finally, *last_part* accounts for the deallocation of the empty input vector.

D.3.14 If

The if function applies its predicate function p to its input value v and, subsequently, produces $c\,v$ if $p\,v = true$ or $p\,a$ if $p\,v = false$.

```
$Str\ \ (\mathsf{if}(p, c, a),\ (s, t, v, o, tr)) = last\_part$
    $\textbf{where}$
    $first\_part \quad = add\_to\_trace\ S(v)\ S(v)\ (s, t, v, o, tr)$
    $(s1, t1, null, b, tr1) \quad = Str(p, first\_part)$
    $last\_part \quad = Str(c, (s1, t1, v, null, tr1)),\ \textbf{if}\, b = true$
                $\quad = Str(a, (s1, t1, v, null, tr1)),\ \textbf{otherwise}$
```

Synopsis *first_part* accounts for the cost of making one extra copy the input value. The middle state, $(s1, t1, i1, b, tr1)$, accounts for the cost of evaluating the predicate function, p. *last_part* simulates the evaluation of either $c\,v$ or $a\,v$.

D.3.15 While

The function $\mathsf{while}(f, p)$ takes an input value, v, and successively applies f to v while $p\, v$ is *true*. If $p\, v = \mathit{false}$ to begin with then f will never be applied. Note that while is potentially non-terminating.

$$
\begin{array}{l}
\mathit{Str}(\ \mathsf{while}(f,p),(s,t,v,o,tr)) = \mathit{cond\_do}\ \mathit{Pval(next\_part)}\ \mathit{first\_part} \\
\quad \boldsymbol{where} \\
\quad \begin{array}{lcl}
\mathit{first\_part} & = & \mathit{add\_to\_trace}\ S(v)\ S(v)\ (s,t,v,o,tr) \\
\mathit{next\_part} & = & \mathit{Str}(p, \mathit{first\_part}) \\
\mathit{cond\_do}\ \mathit{false}\ \mathit{st} & = & \mathit{st} \\
\mathit{cond\_do}\ \mathit{true}\ \mathit{st} & = & \mathit{Str}(\mathsf{while}(f,p),(s3,t3,o3,\mathit{null},tr3))
\end{array} \\
\qquad\qquad \boldsymbol{where} \\
\qquad\qquad (s3,t3,\mathit{null},o3,tr3) = \mathit{Str}(f,\mathit{st})
\end{array}
$$

Synopsis *first_part* accounts for the copying of the input value required to evaluate the predicate, p. *next_part* simulates the application of the predicate function. The output of *next_part* and the state *first_part* are then applied to the local function *cond_do* which either returns the *first_part* or the state produced by a recursive call to the rule for while with a modified state, depending upon whether the result of the predicate is *true* or *false*.

D.3.16 Priffle

The $\mathsf{priffle}$ (predicate-riffle) function take a triple of vectors (bs, xs, ys) and produces a result vector rs which is the result of merging xs and ys according to to the boolean values in bs. More precisely, for each successive b_i, in bs, the next element r_i, in rs, is the next element of xs if $b_i = \mathit{true}$ and the next element of ys otherwise. Note that $\#bs = \#rs = \#xs + \#ys$.

$$
\begin{aligned}
Str(\ \ & \mathsf{Priffle}, (s, t, (\ [b_0, ..., b_{n-1}],\\
& \qquad\qquad [x_0, ..., x_{l-1}],\\
& \qquad\qquad [y_0, ..., y_{m-1}]), o, tr)) = \mathit{last\_part}
\end{aligned}
$$

where

$$
\begin{aligned}
\mathit{first\_part} &= \mathit{add\_to\_trace}\ (s, t, ([b_0, ..., b_{n-1}], xs, ys), o, tr)\ 1\ 1\\
\mathit{main\_part} &= \mathit{pass\_between}\ [f_0, ..., f_{n-1}]\ \mathit{first\_part}\\
\mathit{last\_part} &= \mathit{add\_to\_trace\_null}\ \mathit{main\_part}\ -3\ 3
\end{aligned}
$$

$$
\begin{aligned}
f_i\quad (s, t, (\ & [b_i, ..., b_{n-1}], [x_j, ..., x_{l-1}], [y_k, ..., y_{m-1}]),\\
& [r_0, ..., r_{i-1}], tr) = (s, t1, v1, o1, tr1)
\end{aligned}
$$

where

$$
\begin{aligned}
v1 &= ([b_{i+1}, ..., b_{n-1}], [x_{j+1}, ..., x_{l-1}], [y_k, ..., y_{m-1}]), \ \textbf{\textit{if}}\ b_i = \mathit{true}\\
&= ([b_{i+1}, ..., b_{n-1}], [x_j, ..., x_{l-1}], [y_{k+1}, ..., y_{m-1}]), \ \textbf{\textit{otherwise}}\\
o1 &= [r_0, ..., r_{i-1}, x_j], \ \textbf{\textit{if}}\ b_i = \mathit{true}\\
&= [r_0, ..., r_{i-1}, y_k], \ \textbf{\textit{otherwise}}\\
tr1 &= \mathit{add\_to\_trace}\ tr\ 0\ t1\\
t1 &= S(\mathit{last}\, o1) - S(b_i)
\end{aligned}
$$

Synopsis *first_part* accounts for the allocation of a new result vector. *main_part* simulates the transfer of values from either xs or ys according to the next value in bs. Space for new output values is allocated and space consumed by the input values used is deallocated. Finally, *last_part* accounts for the deallocation of the three, now-empty, input vectors.

D.3.17 Mask

mask takes a pair (xs, bs) and filters xs according to the boolean values in bs. If $bs_i = \mathit{true}$ then xs_i is included in the result vector, otherwise it isn't.

mask is used, in conjunction with priffle, to process the functions, corresponding to different branches of a conditional, embedded in a map body, separately.

```
Str( mask, $(s, t, (xs, [b_0, ..., b_{n-1}]), o, tr)) = last\_part$
    where
    $first\_part \; = \; add\_to\_trace\ (s, t, (xs, [b_0, ..., b_{n-1}]), null, tr)\ 1\ 1$
    $main\_part \; = \; pass\_between\ [f_0, ..., f_{n-1}]\ first\_part$
    $last\_part \; = \; add\_to\_trace\_null\ main\_part\ -(S(xs))\ 1$
    $f_i\ (s, t, \; (xs, [b_i, ..., b_{n-1}]), os, tr) =$
            $(s1, t1, (xs, [b_{i+1}, ..., b_{n-1}]), os +\!\!+\, o, tr1)$
            where
            $o \; = \; [xs!i],\ \textbf{if}\ b_i$
              $= \; [\,],\ \textbf{otherwise}$
            $s1 \; = \; s + (S(o) - 1) - S(b_i)$
            $t1 \; = \; t + (S(o) - 1) + 1$
            $tr1 \; = \; tr +\!\!+ (s1, t1)$
```

Synopsis *first_part* accounts for the allocation of a new empty vector for the result. *main_part* simulates the conditional copying of elements of xs according to the corresponding values in bs. Elements of both input vectors are deallocated as each f_i, called to produce *main_part*, is invoked. Finally, *last_part* accounts for the deallocation of both, now-empty, input vectors.

D.3.18 Transpose

In the current time-space model transpose is specialised to operate on just the outer two dimensions of the nested input vector. That is, in this model,

$$\mathsf{transpose} = \mathsf{transpose}_{(0,1)}$$

For maximum flexibility, future versions of this model must be extended to implement transpose over arbitrary dimensions in the same fashion as the model described in appendix B. The current rule for transpose is:

$$
\begin{array}{l}
Str(\ \textsf{transpose}, (s,t,xs,o,tr)) = last\_part \\
\quad \textbf{\textit{where}} \\
\quad \begin{array}{lcl}
last\_part & = & (s2, t2, null, newxs, tr2) \\
s2 & = & s1 - S(xs) \\
t2 & = & t2 + 1 \\
tr2 & = & tr1 +\!\!+ (s2, t2) \\
s1 & = & s + S(newxs) \\
t1 & = & t + S(newxs) \\
tr1 & = & tr +\!\!+ (s1, t1) \\
newxs & = & trans\ xs
\end{array}
\end{array}
$$

$$
trans\ \begin{array}{l}
[[x_{(0,0)}, x_{(0,1)}, \ldots, x_{(0,m_0-1)}], \\
\ [x_{(1,0)}, x_{(1,1)}, \ldots, x_{(1,m_1-1)}], \\
\qquad\qquad \vdots \\
\ [x_{(n-1,0)}, x_{(1,1)}, \ldots, x_{(n-1,m_{n-1}-1)}]]
\end{array}
=
\begin{array}{l}
[[x_{(0,0)}, x_{(1,0)}, \ldots, x_{(n-1,0)}], \\
\ [x_{(0,1)}, x_{(1,1)}, \ldots, x_{(n-1,1)}], \\
\qquad\qquad \vdots \\
\ [x_{(0,m_0-1)}, x_{(1,m_0-1)}, \ldots, x_{(n-1,m_0-1)}]]
\end{array}
$$

Synopsis The trace generated by the rule for transpose abstracts over most internal details of the operation. There are only two steps in the trace. The first step allocates enough space for the transposed result vector. The second step takes time equivalent to the size of this result vector and the final st. The *trans* function, informally, describes the functional semantics of transpose.

This completes the description of the interpreter. Next we, very briefly outline the process we used for extracting and displaying traces.

D.4 Extracting the trace

The interpreter rules described above define the process for generating an interpreter state for a given sequential BMF program and its input data. The process of extracting a trace-graph from this state is straightforward.

First the trace is projected out of the state by using the function:

$$get\_trace\ (s,t,i,o,tr) = tr$$

the trace *tr* is in the form of a list of pairs. For plotting this list is converted into a text file with a pair of numbers on each line. This is done using the Miranda `show` function and facilities for file output.

Once the trace data is saved to a file in the above format it is a simple matter to plot the data in a graphing package. In our case we used *Gnuplot*.

Appendix E

Nested Parallelism

Thus far, this report has described the introduction of one level of parallelism to BMF code. While one level is satisfactory for many applications there are some conditions under which it is desirable to exploit parallelism on more than one level by nesting parallel constructs. For the Adl programmer there are three broad contexts in which the use of nested parallelism may be warranted by the improved performance it generates:

- The extension of block partitioning to more than one dimension;
- the efficent exploitation of parallelism on irregular nested vectors with fixed shape;
- and the efficient exploitation of parallelism on irregular nested vectors with dynamically changing shape.

In the following, the use of nested parallelism in each of these three contexts is discussed in turn. First however, as a preparatory step, we present the methodology we use for expressing nested parallelism in BMF code.

E.1 Methodology

For most functions, adding nested parallelism to BMF code is a simple extension of the process for adding a single layer of parallelism to BMF code. Figure 174 shows the process of propagating a single layer of parallelism through a BMF program which calculates the sums of squares of an input vector. The outermost function types

$$\begin{array}{lr}
+/_{([int]\to int)} \cdot (\times \cdot (\mathsf{id},\mathsf{id})^\circ)*_{([int]\to[int])} & \Rightarrow \\
\mathsf{id}_{(int\to int)} \cdot +/_{([int]\to int)} \cdot (\times \cdot (\mathsf{id},\mathsf{id})^\circ)*_{([int]\to[int])} & \Rightarrow \\
+/^{\|}_{([int]^{\|}\to int)} \cdot (+/) *^{\|}_{([[int]]^{\|}\to[int]^{\|})} \cdot \mathsf{split}_{p\,([int]\to[[int]]^{\|})} \cdot (\times \cdot (\mathsf{id},\mathsf{id})^\circ)*_{([int]\to[int])} & \Rightarrow \\
+/^{\|}_{([int]^{\|}\to int)} \cdot (+/) *^{\|}_{([[int]]^{\|}\to[int]^{\|})} \cdot (\times \cdot (\mathsf{id},\mathsf{id})^\circ) * *^{\|}_{([[int]]^{\|}\to[[int]]^{\|})} \cdot \mathsf{split}_{p\,([int]\to[[int]]^{\|})} &
\end{array}$$

Figure 174. Parallelisation process for code to calculate the sum of squares for a list of numbers.

$$\begin{array}{lr}
+/^{\|} \cdot (+/) *^{\|} \cdot (\times \cdot (\mathsf{id},\mathsf{id})^\circ) * *^{\|} \cdot \mathsf{split}_p & \Rightarrow \\
+/^{\|} \cdot \mathsf{id} \cdot (+/) *^{\|} \cdot (\times \cdot (\mathsf{id},\mathsf{id})^\circ) * *^{\|} \cdot \mathsf{split}_p & \Rightarrow \\
+/^{\|} \cdot (+/^{\|} \cdot (+/)*^{\|}) *^{\|} \cdot \mathsf{split}_q *^{\|} \cdot (\times \cdot (\mathsf{id},\mathsf{id})^\circ) * *^{\|} \cdot \mathsf{split}_p & \Rightarrow \\
+/^{\|} \cdot (+/^{\|} \cdot (+/)*^{\|}) *^{\|} \cdot (\times \cdot (\mathsf{id},\mathsf{id})^\circ) * *^{\|} *^{\|} \cdot \mathsf{split}_q \cdot \mathsf{split}_p & = \\
+/^{\|} \cdot (+/^{\|}) *^{\|} \cdot (+/) *^{\|} *^{\|} \cdot (\times \cdot (\mathsf{id},\mathsf{id})^\circ) * *^{\|} *^{\|} \cdot \mathsf{split}_q \cdot \mathsf{split}_p & \Rightarrow
\end{array}$$

Figure 175. Propagation of a second layer of parallelism through the program from figure 174.

in the program are written as subscripts. Parallel vector types are annotated with the $(\|)$ superscript. Another layer of parallelism can be added to this program by propagating parallel functions through the sequential code embedded inside parallel constructs. In general, any function f whose type unifies with $[[\alpha]]^{\|} \to \beta$ is a candidate for adding another layer of parallelism. The most downstream candidate function in the parallelised program in figure 174 is $(+/)*^{\|}_{([[int]]^{\|}\to[int]^{\|})}$ and this is where the process of propagating a second layer of parallelism starts in figure 175[1]. Note that the transformation between the last two lines does not serve to propagate parallelism any further into the code, but rather it distributes the outer map function in $(+/^{\|} \cdot (+/)*^{\|})*^{\|}$ over its component functions in order to more clearly demarcate the stages of the program. The nested parallel program in the last line of figure 175 introduces a hierarchy of parallelism where each element of the input vector is distributed once by split_p and each of these distributed elements is further distributed by $\mathsf{split}_q*^{\|}$. This hierarchy is apparent in the snapshots, shown in figure 176, of the execution of the nested parallel program on the last line of figure 175. Note that, in this case, the hierarchy is balanced which makes it a simple matter to map the virtual nodes of the hierarchy efficiently onto actual nodes of a machine[2]. In

[1]The type subscripts have been omitted for the sake of readability.

[2]The simulator, described in chapter 6, uses the mapping:

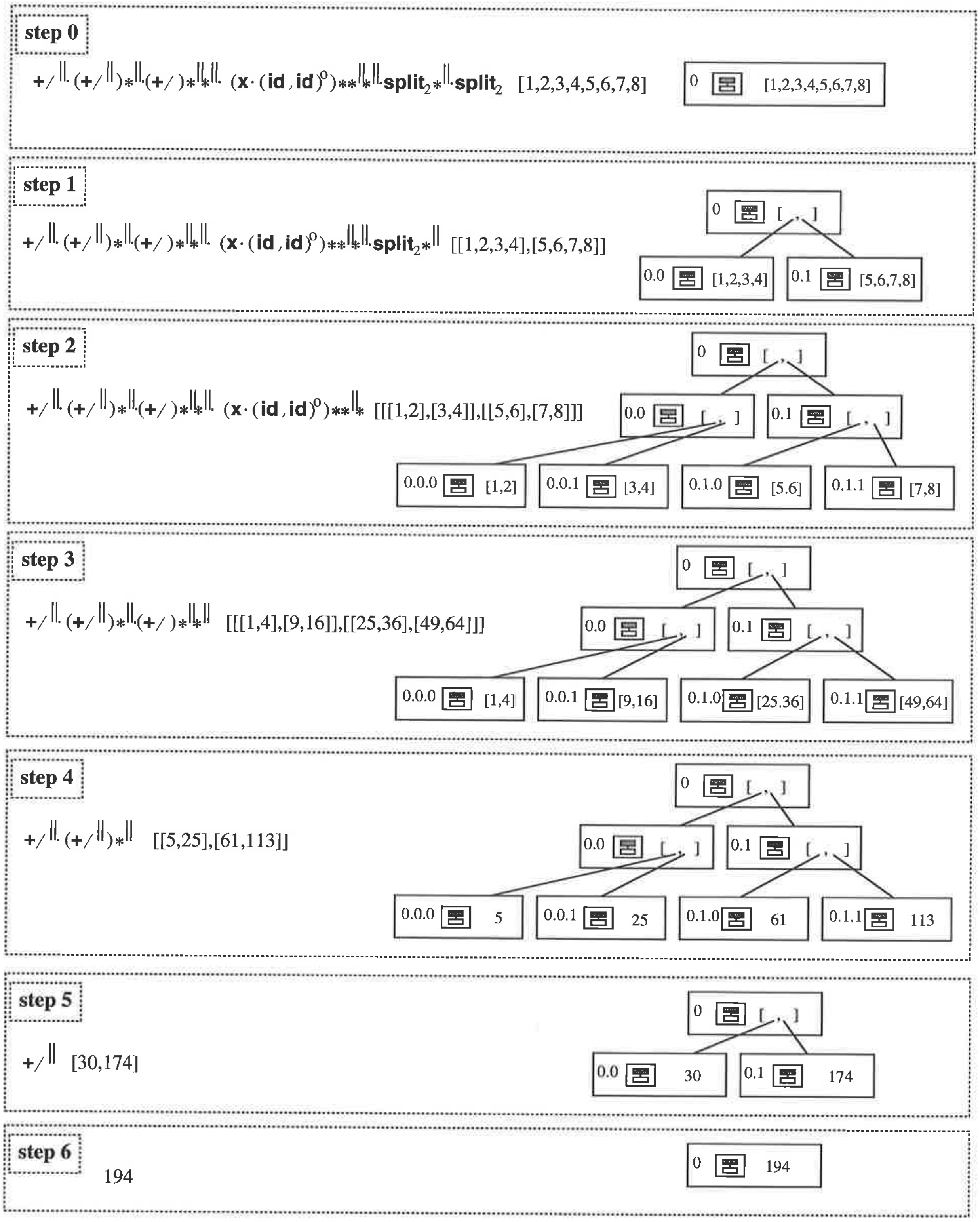


Figure 176. Snapshots of the execution of the nested parallel program from figure 175.

general, hierarchies do not have to be balanced and, in programs with the appropriate conditional functions, they do not even have to have uniform depth. Such unbalanced hierarchies will require a more sophisticated mapping from virtual to real nodes.

It should also be noted that applying nested parallelism to the one-dimensional sum-of-squares program above is not going to produce better performance than applying the same degree of parallelism on a single level. However, there are general classes of problem where nested parallelism can be of substantial benefit and the remainder of this chapter is devoted to describing these classes of problem and outlining how nested parallelism can be applied to them. We start with problems that benefit from the extension of block partitioning to more than one dimension.

E.2 Using nesting to extend block partitioning to more than one dimension

In Adl, and in its implementation, multidimensional vectors are emulated by nested single-dimensional vectors. So, for example, the two dimensional structure in figure 177(a) is emulated, in Adl, by the nested vector in figure 177(b). Partitioning nested vectors along one dimension is straightforward: figure 178 shows a simple partitioning of the outer dimension of the nested vector shown in figure 177(b). This partitioning, which may be appropriate if the application requires limited interaction between elements of sub-vectors, divides the array into longitudinal strips. Unfortunately, there are many applications that require substantial interactions spanning more than one dimension. Stencil algorithms, for example, typically require interactions between each element and its neighbouring elements in all dimensions. Figure 179(a) shows the inter-node interactions generated a stencil algorithm running over the partitioned vector from figure 178. Figure 179(b) shows the communications arising from an alternative partitioning of the same vector. A simple count of the number of arrows shows that the two-dimensional block-partitioning shown in (b) generates less communication than the one-dimensional block partitioning. If we

$$\{[0, 0.0, 0.0.0] \mapsto \textit{node 0}, [0.0.1] \mapsto \textit{node 1}, [0.1, 0.1.0] \mapsto \textit{node 2}, [0.1.1] \mapsto \textit{node 3}\}$$

of nodes of the virtual machine to physical nodes. Note that this mapping places virtual parent nodes on the same physical nodes as their children. This is done on the assumption that the amount of work done by parent virtual nodes is small compared to the work done by virtual nodes at the bottom of the hierarchy.

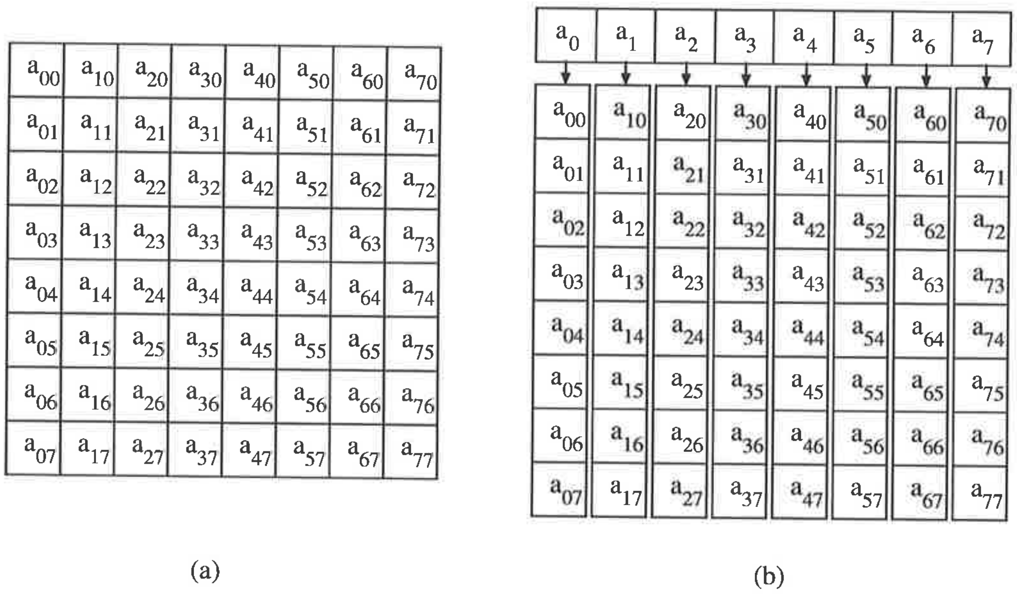


Figure 177. A two dimensional structure, part (a), and its representation in Adl, part (b).

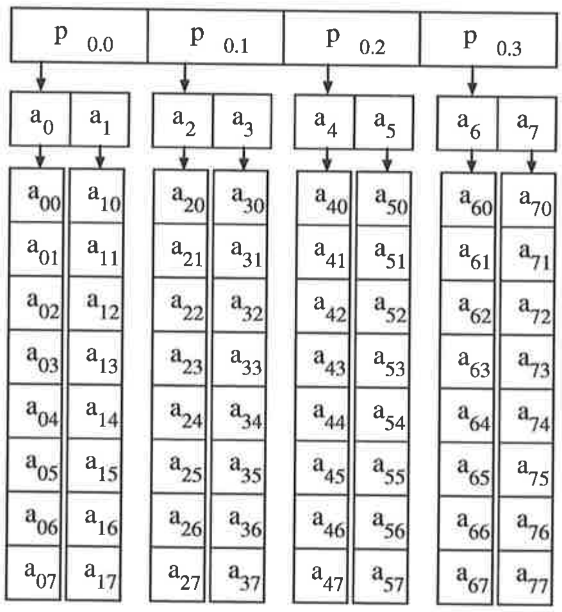


Figure 178. Simple 4-way partitioning of the outer dimension of the nested vector shown in figure 177(b).

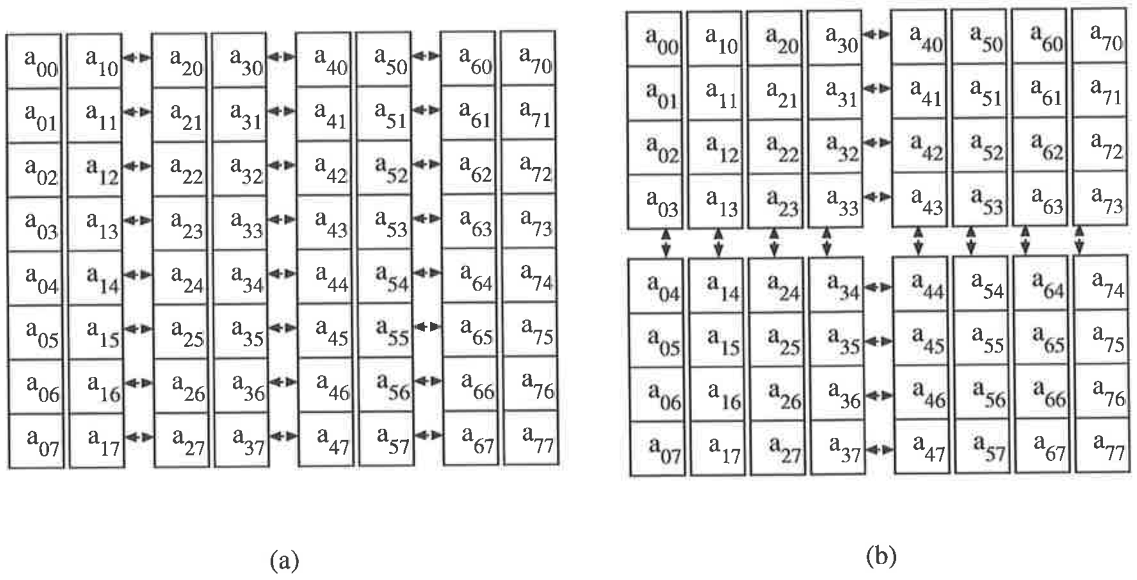

a00 a10 a20 a30 a40 a50 a60 a70
a01 a11 a21 a31 a41 a51 a61 a71
a02 a12 a22 a32 a42 a52 a62 a72
a03 a13 a23 a33 a43 a53 a63 a73
a04 a14 a24 a34 a44 a54 a64 a74
a05 a15 a25 a35 a45 a55 a65 a75
a06 a16 a26 a36 a46 a56 a66 a76
a07 a17 a27 a37 a47 a57 a67 a77
(a)
(b)

Figure 179. Comparison of communications costs arising from a stencil algorithm on a one-dimensional partitioning of a two dimensional vector, part (a), and a two dimensional partitioning of the same vector, part (b).

generalise the partitioning schemes in part (a) and (b) to p nodes and square datasets whose side is length n we find that the partitioning scheme of part (a) gives rise to:

$$(p-1)n$$

two-way inter-node communications and the partitioning scheme of part (b) gives rise to:

$$2(\sqrt{p}-1)n$$

two-way inter-node communications. We can see that, in terms of communications, a two-dimensional block partitioning of data for a two dimensional stencil algorithm is asymptotically better than a one dimensional partitioning. This means that, for some important algorithms, there are substantial gains to be had by dividing our data in more than one dimension. In Adl this means that it advantageous to exploit parallelism both between and within sub-vectors, which entails the use of nested parallelism.

E.2.1 Nested block partitioning in Adl

Nested parallelism on nested structures in Adl is implemented by propagating parallel constructs through the outer and inner level of the array. To illustrate this process,

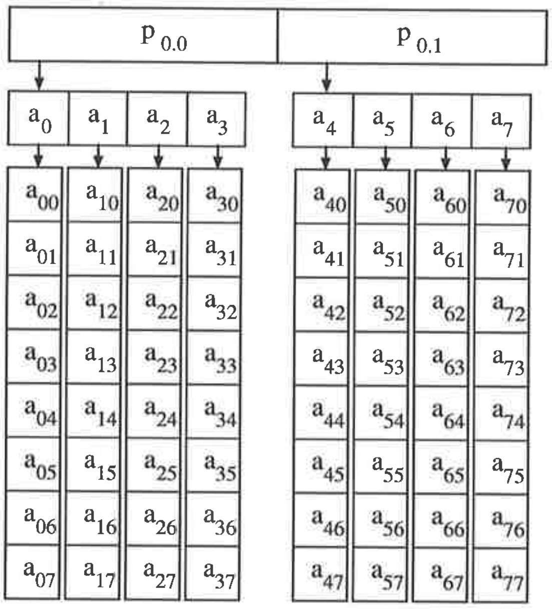

p 0.0
p 0.1
a0 a1 a2 a3
a4 a5 a6 a7
a00 a10 a20 a30 a40 a50 a60 a70
a01 a11 a21 a31 a41 a51 a61 a71
a02 a12 a22 a32 a42 a52 a62 a72
a03 a13 a23 a33 a43 a53 a63 a73
a04 a14 a24 a34 a44 a54 a64 a74
a05 a15 a25 a35 a45 a55 a65 a75
a06 a16 a26 a36 a46 a56 a66 a76
a07 a17 a27 a37 a47 a57 a67 a77

Figure 180. Nested vector from figure 177(b) partitioned across two nodes in the outer dimension.

imagine a sequential BMF program working over a nested vector:

$$f**_{[[\alpha]]\to[[\beta]]}$$

such a program would accept input, and produce output, with a structure similar to that shown in figure 177(b). The first step of parallelisation propagates split through $f**$ giving:

$$+\!\!+/^{\|}_{[[[\beta]]]^{\|}\to[[\beta]]} \cdot f***_{[[[\alpha]]]^{\|}\to[[[\beta]]]^{\|}} \cdot \mathsf{split}_{2\,[[\alpha]]\to[[[\alpha]]]^{\|}}$$

In this version, the function $f***^{\|}$ is presented with data with a structure like that shown in figure 180. The second stage of parallelisation propagates split through $f*$ the innermost map of $f***^{\|}$ giving:

$$+\!\!+/^{\|}_{[[[\beta]]]^{\|}\to[[\beta]]} \cdot (+\!\!+/^{\|}_{[[\beta]]^{\|}\to[\beta]} \cdot f**^{\|}_{[[\alpha]]^{\|}} \cdot \mathsf{split}_{2\,[\alpha]\to[[\alpha]]}) **^{\|}_{[[[\alpha]]]^{\|}\to[[[\beta]]]^{\|}} \cdot \mathsf{split}_{2\,[[\alpha]]\to[[[\alpha]]]^{\|}}$$

we can further transform the program above to consolidate the splitting and merging into distinct phases[3]:

$$+\!\!+/^{\|} \cdot (+\!\!+/^{\|}) **^{\|} \cdot f**^{\|}**^{\|} \cdot \mathsf{split}_2 **^{\|} \cdot \mathsf{split}_2$$

The central $f**^{\|}**^{\|}$ function is presented with data having the nesting structure shown in figure 181. This partitioning allocates a square-shaped block to each of the

[3]Again, we have omitted type subscripts for the sake of readability.

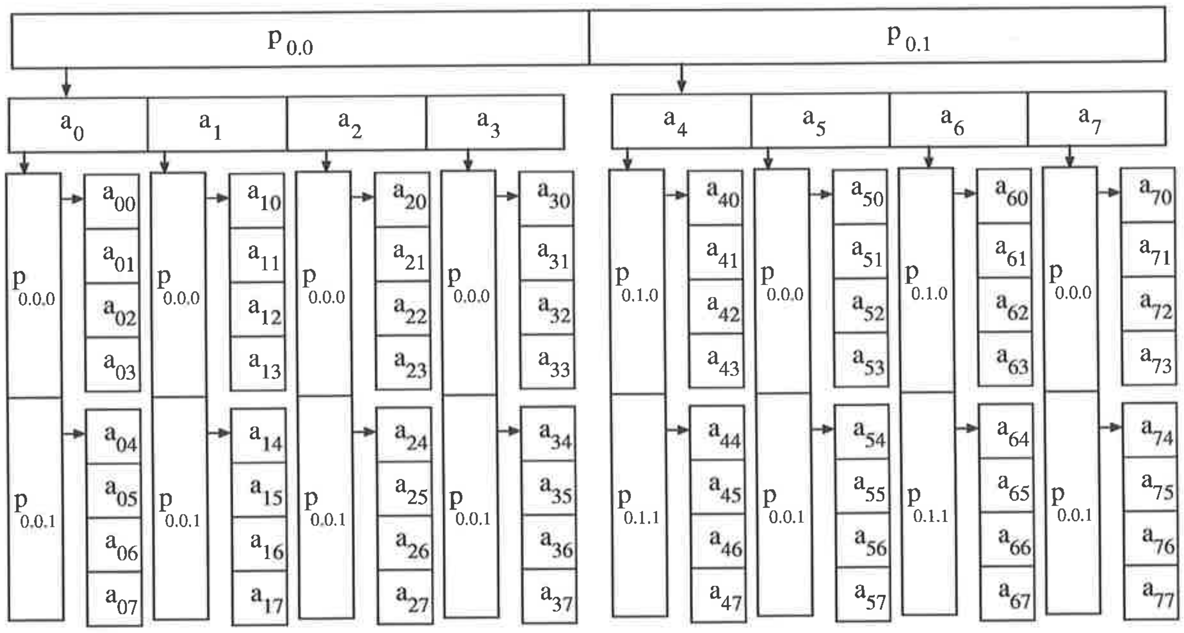

p0.0
p0.1
a0
a1
a2
a3
a4
a5
a6
a7
p0.0.0
p0.0.1
p0.1.0
p0.1.1
a00 a01 a02 a03 a04 a05 a06 a07
a10 a11 a12 a13 a14 a15 a16 a17
a20 a21 a22 a23 a24 a25 a26 a27
a30 a31 a32 a33 a34 a35 a36 a37
a40 a41 a42 a43 a44 a45 a46 a47
a50 a51 a52 a53 a54 a55 a56 a57
a60 a61 a62 a63 a64 a65 a66 a67
a70 a71 a72 a73 a74 a75 a76 a77

Figure 181. The nested vector from figure 177(b), block-partitioned in both the outer and inner dimensions.

physical nodes mapped to the virtual nodes 0.0.0, 0.0.1, 0.1.0 and 0.1.1. The inner function f is applied simultaneously in each of these nodes in the sequence indicated by the four thick arrows in figure 182. Note that the innermost level of parallel execution occurs inside a sequential map. To date, in this work, we have implicitly assumed that in an application of sequential map:

$$(f) * [x_0, x_1, \ldots, x_{n-1}] = [f\, x_0, f\, x_1, \ldots, f\, x_{n-1}]$$

the execution of each application, $f\, x_i$ must finish completely before execution the next application $f\, x_{i+1}$ begins. This makes sense when each application, itself, runs sequentially. However, in the execution trace represented in figure 182 the function $f{*}{*}^{\|}$, applied to the members of the sequential vectors $[a_0, a_1, a_2, a_3]$ and $[a_4, a_5, a_6, a_7]$, is parallel. If we insist, for example, that $f * *^{\|}\, a_0$ finish completely before we start the execution of any of $f * *^{\|}\, a_1$ then we introduce an unnecessary synchronisation between nodes 0.0.0 and 0.0.1. In order to avoid the costs of such synchronisations the semantics of sequential map should be refined to have explicit synchronisation only at the end of the map in cases where the map-body is parallel.

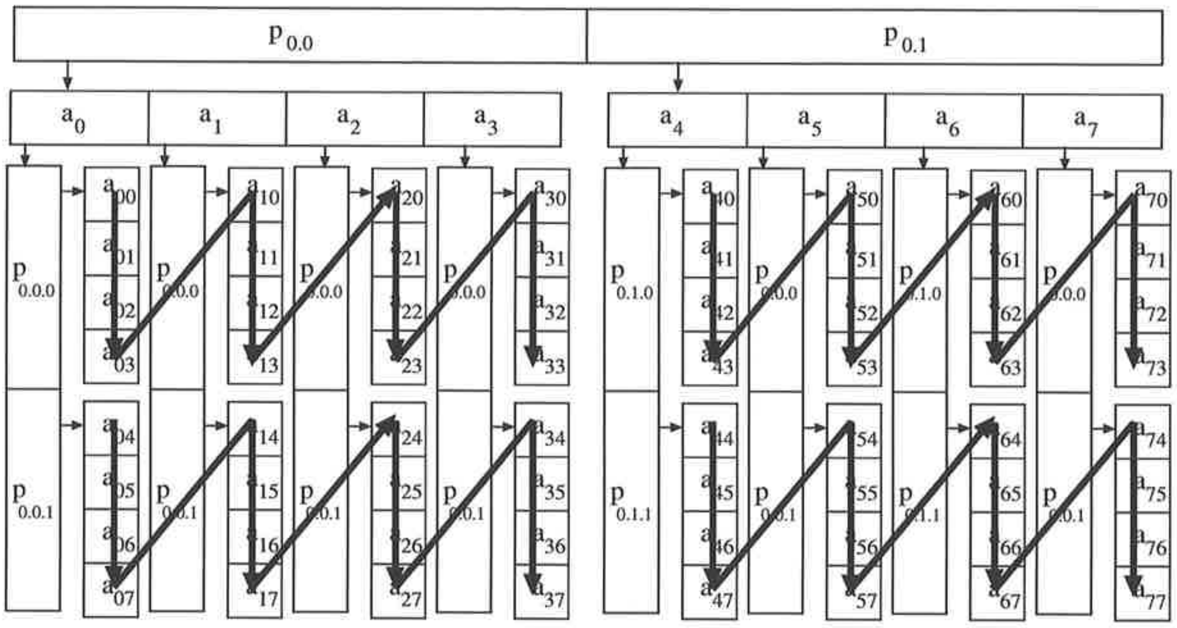


Figure 182. Trace of execution of $f **^{\|} **^{\|}$ applied to the data structure from figure 181.

E.2.2 Nesting other structures

So far we have looked at how to propagate nested parallelism through the BMF code produced by the Adl implementation in order to partition data into blocks in more than one dimension. For most BMF functions this process is a straightforward extension of the single-level parallelisation process. However, some important constructs, such as select, defy a simple extension of parallelisation strategies to a nested context. This is unfortunate because important classes of parallel algorithm, such as stencil algorithms make intensive use of select. For example, a sequential version of a simple stencil that repeatedly applies a binary operator f to each element and its lower-left neighbour until a condition is met can be written:

$$\mathsf{while}(f ** \cdot \mathsf{zip} \cdot (\mathsf{id}, \mathsf{select} \cdot (\mathsf{id}, \mathit{lshift})^{\circ} \cdot \mathsf{select} * \cdot \mathsf{zip} \cdot (\mathsf{id}*, \mathit{ushift})^{\circ})^{\circ}, \mathit{not\_done})$$

Where *lshift*, *ushift*, and *not_done* are place-holders for BMF code to generate left-shifted indices, generate up-shifted indices and perform a boolean test for loop completion respectively. Parallelisation of the program above in one dimension mostly involves replacing the left-most select with distselect composed with some additional detailed code as per the rules in chapter 6. However, parallelising two dimensions to achieve a block partitioning leads to a much more complex code, involving at least:

1. A stage to set up data for inter-block transfers.
2. A stage to perform the inter-block transfers.
3. A stage to perform intra-block transfers.

In general, each stage will involve calls to select and/or distselect as well as detailed code to create the index vectors for these calls. The level of detail involved is somewhat at odds with the relatively simple operation being performed.

E.2.3 Simpler block-partitioning

A strong argument could be made that part of the complexity arising from block partitioning is due to the emulation of multi-dimensional arrays using nested lists. If Adl supported genuine multi-dimensional arrays there would be no need for such emulation. With multi-dimensional arrays as a primitive type, functions such as map could be naturally applied to all dimensions and block partitioning could be specified without the need for nesting. The expression of parallelism over multiple-dimensions of regular multi-dimensional structures would be straightforward.

Such an extension to the Adl type system would involve the redefinition of vector primitives such as map, reduce, scan, iota, length and index in a generic, *shape-polymorphic* way. Such polymorphism has already been extensively explored in languages such as FISh[85] and in a parallel-context in GoldFISh[86] so there are already solid foundations for such an approach. However, the use of multi-dimensional arrays only supplants the need for nested parallelism when structures are regular. Nested parallelism becomes more attractive when a problem does not lend itself to an efficient solution using regular structures. We examine strategies for exploiting nested parallelism over irregular structures in the next two sections.

E.3 Using nested parallelism to exploit parallelism over static irregular structures

Many parallel applications, including some finite-element and sparse-matrix algorithms, operate over static data structures whose optimal distribution does not evolve significantly over the course of the computation. In these cases, it is appropriate

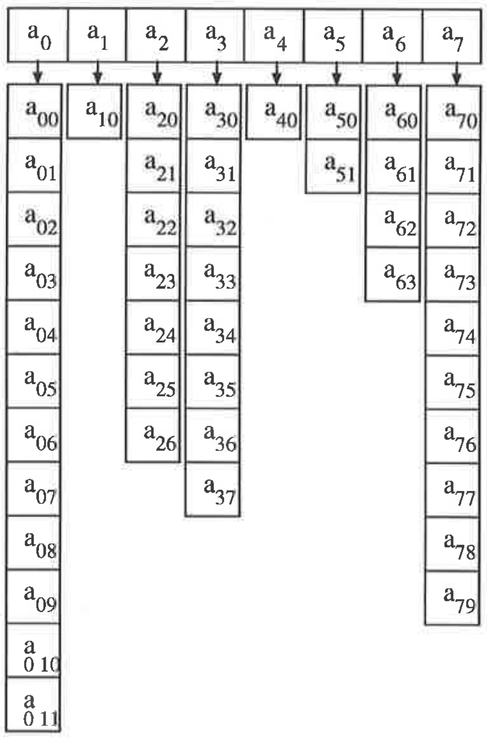


Figure 183. An irregular nested vector

to exploit parallelism using a static nested decomposition of irregular data over the nodes of the machine.

In Adl irregular data structures are best expressed using nested vectors. The effective exploitation of parallelism over such vectors will involve nested parallelism. In the last section, we showed how multiple levels of nesting could be used to produce a block partitioning. The same techniques could be easily applied to irregular nested structures but, without modification, would be prone to produce poor load balancing. In this section, we show how the techniques from the last section can be modified to produce better load-balancing.

E.3.1 A simple schema

Figure 183 shows an irregular nested vector which is the intended data-source for a parallel computation. Figure 184 shows a one dimensional decomposition of the vector. Assuming that each data value corresponds to a similar amount of

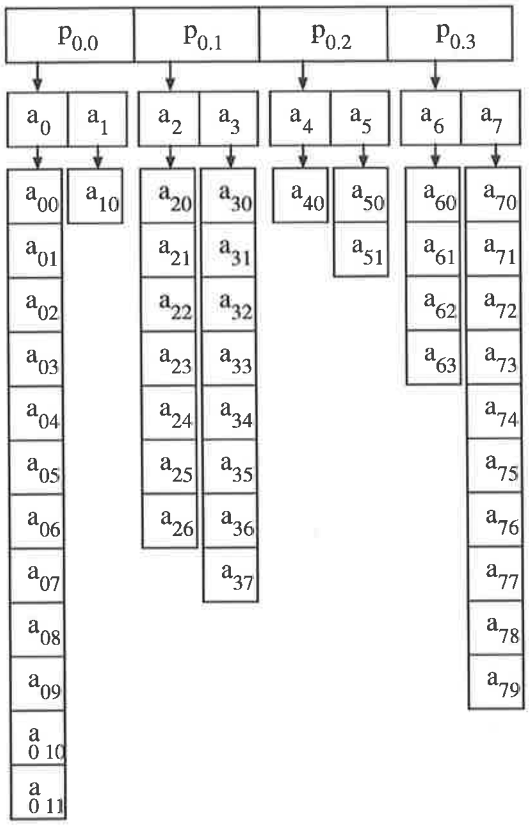

$p_{0.0}$
$p_{0.1}$
$p_{0.2}$
$p_{0.3}$
a_0 a_1 a_2 a_3 a_4 a_5 a_6 a_7
a_{00} a_{01} a_{02} a_{03} a_{04} a_{05} a_{06} a_{07} a_{08} a_{09} $a_{0\,10}$ $a_{0\,11}$
a_{10}
a_{20} a_{21} a_{22} a_{23} a_{24} a_{25} a_{26}
a_{30} a_{31} a_{32} a_{33} a_{34} a_{35} a_{36} a_{37}
a_{40}
a_{50} a_{51}
a_{60} a_{61} a_{62} a_{63}
a_{70} a_{71} a_{72} a_{73} a_{74} a_{75} a_{76} a_{77} a_{78} a_{79}

Figure 184. Irregular nested vector from figure 183 partitioned in one dimension, producing poor load-balancing.

computation[4], this regular decomposition gives a poor load-balance[5]. If we are prepared to use more processing nodes we can achieve a better load-balance by exploiting nested parallelism. Figure 185 shows how each of the larger sub-vectors can be further distributed to achieve a more even global distribution of work. Note that each sub-vector resides on at least one separate node and that the longer sub-vectors are divided over different numbers of nodes to the shorter sub-vectors. This non-uniformity in node-allocation requires the use of a split that is sensitive to the statically determined partitioning scheme for its data. This means that instead of having:

$$\mathsf{split}_p * \cdot \mathsf{split}_q$$

[4]For most computations, this assumption is very reasonable but there are exceptions, such as LU-factorisation of matrices, so the assumption is worth making explicit.

[5]Even if we are prepared to use contiguous partitions with different numbers of elements in each a better load balance can be obtained but it is possible, for a given number of nodes, to come up with a data distribution that defies a load-balanced one-dimensional decomposition.

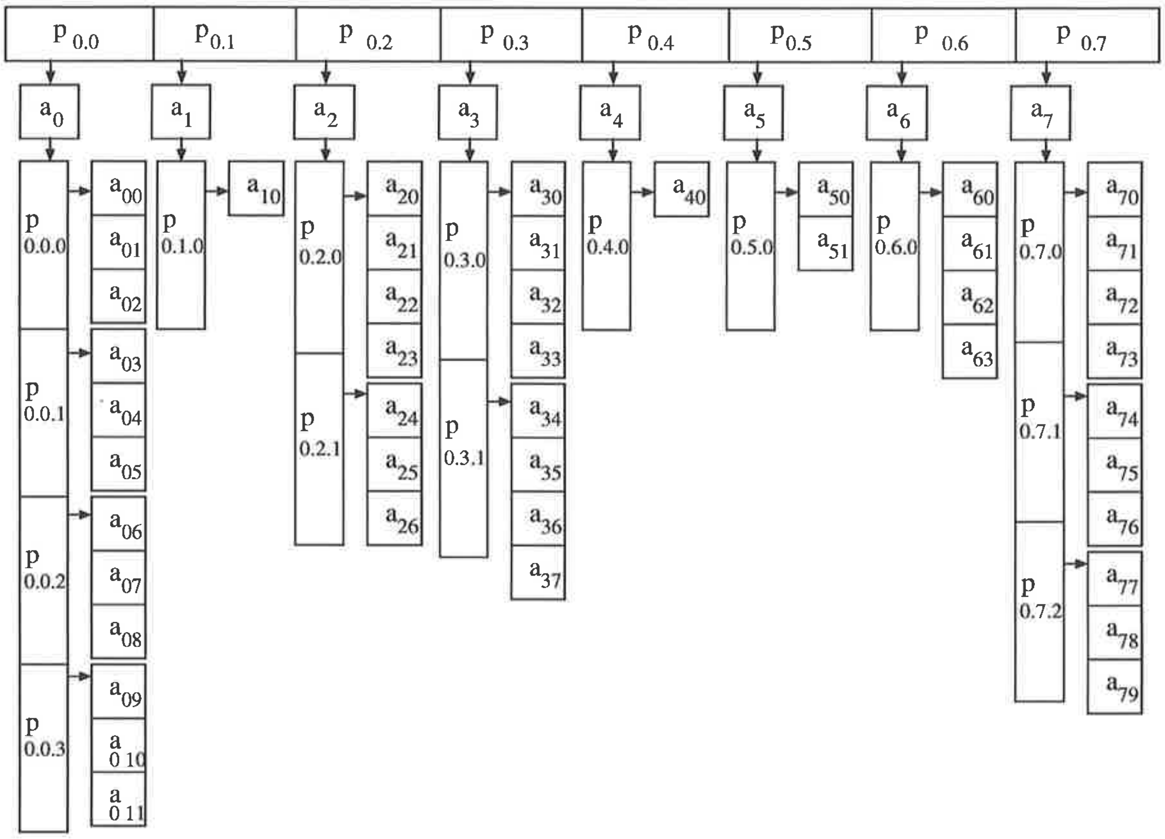


Figure 185. Irregular nested vector from figure 183 partitioned in two dimensions, producing improved load-balancing.

as used in the last section, to sub-divide data, we use:

$$\mathsf{split}_{p_i} * \cdot \mathsf{split}_q$$

where each p_i is determined by the partitioning function for the input data. The partitioning scheme shown in figure 185 is is good but not perfect. Nodes $p_{0.1.0}$, $p_{0.4.0}$ and $p_{0.5.0}$ have a small amount of work compared to the others. These small chunks of work can be consolidated by choosing an appropriate mapping of virtual nodes to physical nodes. For example, a mapping:

$$\{\ [0, 0.0, 0.0.0] \mapsto \textit{Node 0}, [0.0.1] \mapsto \textit{Node 1}, [0.0.2] \mapsto \textit{Node 2},$$
$$[0.0.3, 0.1, 0.1.0] \mapsto \textit{Node 3}, [0.2, 0.2.0] \mapsto \textit{Node 4}, [0.2.1] \mapsto \textit{Node 5},$$
$$[0.4, 0.4.0, 0.5, 0.5.0] \mapsto \textit{Node 6}, [0.6, 0.6.0] \mapsto \textit{Node 7}, [0.7, 0.7.0] \mapsto \textit{Node 8},$$
$$[0.7.1] \mapsto \textit{Node 9}, [0.7.2] \mapsto \textit{Node 10}\}$$

will achieve a reasonable load-balance at the cost of run-time infrastructure to support the mapping[6].

E.3.2 Nesting other functions in an irregular context

The issues that arise from nesting other structures besides map in an irregular context are similar to those in a regular context. For most functions, the propagation of a nested level of parallelism through the program is a simple extension of propagating a single level of parallelism. However, some constructs, such as select produce quite detailed code when parallelised in more than one dimension. With irregular structures, the optimal implementation of the partitioning function is different from that in a regular context. In a regular context the partitioning function can, typically, be captured as a simple mathematical function. In an irregular context, the partitioning function is much more likely to be implemented as an index into a lookup table[7]. The overhead of maintaining such a table is minimal in most applications if the partitioning is static. This overhead grows if the partitioning of data structures evolve as computation proceeds. We examine the issues surrounding such dynamically evolving nested parallelism next.

E.4 Using nested parallelism to exploit parallelism over evolving irregular structures.

Important classes of parallel application including of molecular, gravitational, and weather simulations, create an uneven and evolving distribution of work over their computational domain. In such applications, any static distribution of work in a parallel machine is likely to produce a poor load-balance at some point in the computation. In response to this problem, application writers have employed strategies to, periodically, migrate work from busy nodes to idle nodes to maintain a reasonable load-balance. These strategies fall into two broad categories:

[6]It should be noted that having multiple virtual processors per-node introduces a higher-than-necessary communications costs for efficient parallel implementations of algorithms such as scan. However, if the number of virtual processors per node is kept at a moderate level then these extra costs are not prohibitive.

[7]The other thread of the Adl project[54] implemented the partitioning function using a highly optimised lookup table.

1. Strategies that maintain the explicit structure of the data and periodically perform explicit migration of work according to various heuristics.
2. Strategies that flatten parallel structures to an single dimension of parallelism and maintain a simple, near optimal, partitioning of data throughout the computation.

Both strategies have their advantages and drawbacks. For example, strategy 2 almost always produces a better load balance than strategy 1 but at the cost of relatively frequent redistribution of data. We briefly examine these strategies in turn.

E.4.1 Using work migration to implement dynamically evolving irregular nested parallelism

Work migration requires some fine-grained control at least at the level of the run-time system over data partitioning. Such fine-grained control could be achieved by having many more virtual processors than physical nodes. A run-time system could then, periodically:

1. migrate data,
2. alter the mappings in the partitioning functions, and
3. broadcast the changes to each physical node.

To preserve correctness these actions would need to be carried out in such a way as to appear atomic from the point of view of the executing program. Other aspects of parallelisation would remain the same as they did in previous sections.

Work migration would impose several overheads on the executing program. These include: the cost of monitoring work balance; the cost of calculating and distributing a new partitioning function; the cost of migrating the data; and the cost of maintaining a large number of virtual processors.

A-priori maintaining the atomicity of work-migration would not be difficult since work-migration could be integrated to parts of the program requiring a global synchronisation.

Note that extent to which load-balancing is maintained is highly dependent on how often work migration occurs, the granularity of virtual processors, and the quality of the information used to inform the work-migration process. It is up to the implementor to carefully balance the costs and benefits of a work-migration strategy. Next, we consider the data-flattening strategy.

E.4.2 Using flattening to implement dynamically evolving irregular nested parallelism

The use of flattening in a parallel functional context was pioneered by Blelloch using the language NESL[22, 23]. The NESL implementation works by flattening nested vectors, maintaining a record of their shape, and using flat segmented data-parallel operations over the flattened data. The segmented operations use the shape descriptors associated with their input vectors to inform the semantics of their operation. Load balancing is preserved by maintaining a simple one-dimensional partitioning of the flattened data. Chakravarty and Keller (and others)[30, 29, 91, 92] in Nepal, their parallel extension to the functional language Haskell, extended the flattening transformation to a much broader variety of types. The also formalised the transformations needed to functions to allow them to work with flattened arguments.

In both cases, the partitioning of data is kept implicit until run-time[8] and the primitives of the language provide the efficient segmented operations needed.

Flattened parallelism could be implemented for Adl by applying a flattening transformation, extending that used in NESL, to allow vectors to contain tuple elements. Such extensions already exist in Nepal so their feasibility is not in question. However, the use of flattening in an Adl implementation would substantially change the compilation process. The usefulness of data-movement-optimisation would be maintained because its transformations would continue to reduce data movement in a flattening implementation. In contrast, the parallelisation process would cease to be a part of the compiler and move to the run-time system. The code-generation process would be less straightforward with the need to map the nested function invocations in BMF code to segmented operations. Finally, it is likely that the run-time system would need to be substantial to help maintain the partitioning as the program runs. In summary, it appears feasible to implement dynamically evolving nested parallelism

[8]Though, it must be noted, that in Nepal the programmer is able to embed sequential lists inside parallel lists.

in Adl using flattening transformations but such an implementation would be very different to the current implementation.

E.5 Conclusion

This chapter has described a methodology for propagating nested parallelism through Adl programs and described three broad classes of application where the exploitation of nested-parallelism is useful. The efficient options for implementation vary from class to class. Regular nested parallelism might be best substituted for a single level of parallelism in the context of a richer type system. Static irregular nested parallelism would typically need a detailed partitioning function and support for more virtual processors. Dynamic irregular nested parallelism will probably need the support of a substantial run-time system to support either work migration or structure flattening.

It seems that the best system to use varies with the regularity of the problem. It seems unlikely that one system will be optimal for all problem domains. Careful consideration of the likely problem domain for Adl is needed before deciding on an implementation strategy.

Appendix F

Costs of `malloc` and `free`

Our parallel simulator, used to generate the results in section 6.4 requires estimates of the cost of `malloc` and `free` in order to model the costs of memory management on each node. In this appendix we describe the outcome of several basic experiments designed to quantify these costs.

F.1 Factors

The costs of `malloc` and `free` depends on a number of factors including:

- the hardware.
- the design of the libraries that implement `malloc` and `free`.
- the quantity and pattern of allocation/deallocation.

These factors interact to produce quite complex patterns of performance that are perhaps best studied empirically. We set up series of basic experiments in an attempt to come up with a simple characterisation of the costs of `free` and `malloc`.

F.2 Methodology

We are interested in the cost of `free` and `malloc` under conditions that we will test in the simulator. These conditions are:

- Frequent calls to malloc for small blocks of data (mostly less than 5000 words).

- A variable but mostly short interval between the allocation of a block with `malloc` and its deallocation using `free`.
- As a corollary to the above two statements we might expect that at a typical point in time dynamic memory might contain a large number of small blocks, adding up to a small to moderate memory consumption overall (at least for the data-sizes used in the simulator).

We want measure costs in terms of instructions since this metric allows reasonable cross-architecture comparisons.

Getting raw times Our raw measurements are performed with high-resolution timers. These timers measure wall-clock time in units of nanoseconds. This resolution is more than adequate for our purposes[1]. Measurements are taken by inserting a call to get the time before and after calls to `malloc` and `free`.

Getting instruction times To convert raw times to instruction times we need an estimate of the cost of basic instructions on each architecture. To do this we created some benchmark programs to measure the cost of varying length sequences of increment[2] and assignment statements. The compiler was run with optimisation turned off to ensure that the code we write is the code that runs. We are searching for the *marginal* cost of instruction execution and this can be extracted and verified by looking at the relative costs of different length instruction sequences.

Overcoming confounding factors There are a number of possible confounding factors including

1. Systematic cache effects caused by calls to the timing functions and the code to record the times.
2. A bias in the results caused by the particular set of block-sizes chosen.
3. A bias in results caused by the particular pattern of allocations and deallocations used.

[1]We might expect the actual resolution of the timers to be somewhat lower and for probe-effects to deliver some randomness in times, though at a scale that is not problematic in an experiment with many trials.

[2]Every increment in each sequence was to the same variable which could limit execution speed on some architectures.

The first factor might be significant if the code collecting the time causes code and data that is useful for `malloc` and `free` to evicted from the caches. We try to minimise this effect by inserting mostly lightweight code between calls to the timers and `malloc` and `free`. It could be argued that some use of the cache is likely to occur as the result of normal execution of program code in any case so, in a sense, the effect of the timing probes might occur in real code anyway.

Factor number two might be a problem if we were unlucky enough to choose block-sizes that, by coincidence, matched points of good or poor performance for `malloc` and `free`. To overcome this we chose a large number of block-sizes (100) at intervals of 50 words (a number of no particular significance on most architectures). Block sizes are always odd (i.e. 1 word, 51 words, 101 words). The randomness of our actual cost measurements seems to indicate that we achieved a reasonably good sample of the spectrum of performance.

Bias resulting from factor number three is difficult to eliminate without knowing, in advance, the likely pattern of memory allocation used by code generator. Our experiments tend to allocate and deallocate data in a systematic way. When we altered an experiment to allocate randomly size blocks of data the peformance data changed but not radically.

Overall, from the experiments we describe below we can expect a reasonable characterisation of the cost of `malloc` and `free`. This characterisation can be improved in a number of ways but, given the variations we tried and the moderate consistency of the actual results under quite different conditions, it is unlikely that our results are far off the mark.

Experimental platforms We timed `malloc` and `free` on two architectures:

- A 4-processor Sun Sparc Ultra running Solaris 5.9.
- A Dual-processor PowerPC running Darwin 6.6

For the first platform we use the `gethrtime()` library call declared in `<sys/time.h>`. For the second platform we used the `mach_absolute_time()` library call declared in `<mach/mach_time.h>`.

F.3 Experimental results

F.3.1 Raw instruction execution speeds

On both platforms we found the marginal cost of increment and assignment instructions were very similar to each other. On both architectures the cost of executing the first instruction in a sequence was very high, very likely, the result of cache-effects. Once the execution of a sequence was underway performance seemed quite consistent. The costs of both assignment and increment statements on the older Sun platform are around 59 nanoseconds (plus or minus one nanosecond). On the newer PowerPC platform the execution costs of increments and assignments are around 0.25 nanoseconds (very small variations around this point).

We have described our experimental setup and some factors to watch out for. Now we can show the results. First we describe the cost of `malloc`.

F.3.2 The cost of `malloc`

In this section we examine the results of a series of experiments to arrive at an estimate of the cost of `malloc`. Each experiment assumes a different pattern of calls to `malloc` and `free`.

Cost of `malloc` without deallocation In our simplest experiment we performed 50 calls to `malloc` on each of a series of 100 block sizes between four bytes (one integer) and 19804 bytes (4951 integers), at 200 byte intervals. In this experiment we made no calls to `free`.

Figures 186 and 187 show scatter-plots of data points collected the Sun and PowerPC platforms respectively. Note the different scales in the diagrams. In both figures the `malloc`'s involving small blocks of data tended to be the least costly[3]. The effect of block-size appears more pronounced on the Sun platform where the cost of allocating larger and larger blocks increases typical costs from around 50 instructions to between 2000 and 10,000 instructions.

For the PowerPC platform costs for small blocks are clustered about two modes at approximately 100 and 3000 instructions respectively. Perversely, average

[3]With the notable exception of the very first `malloc` performed in each program - probably due, in part, to instruction cache effects.

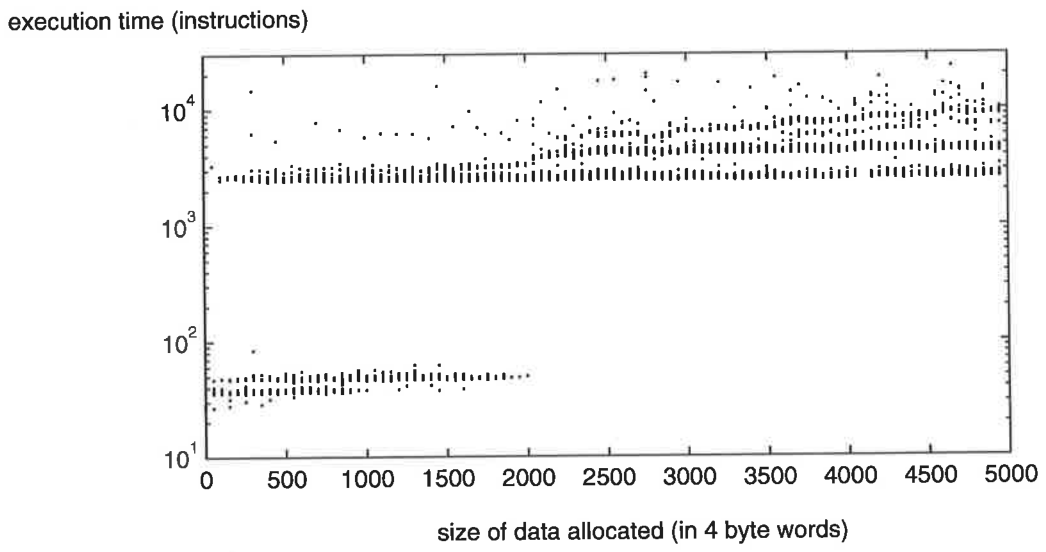


Figure 186. Cost of calls to `malloc`, measured in instructions, on the Sun server platform. 50 calls are made for each block-size and calls are made in ascending order of block-size. No deallocation was performed.

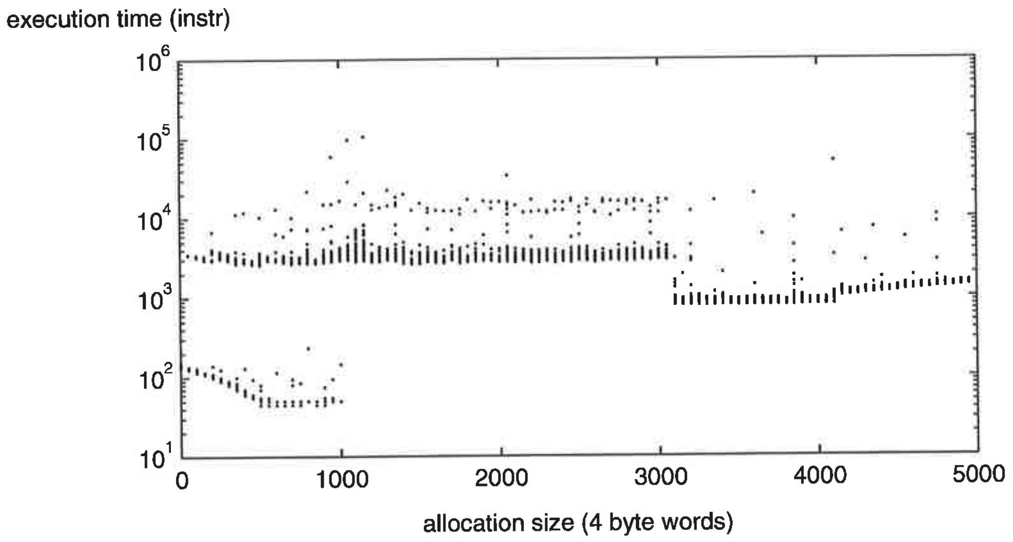


Figure 187. Cost of calls to `malloc`, measured in instructions, on the PowerPC platform. 50 calls are made for each block-size and calls are made in ascending order of block-size. No deallocation was performed.

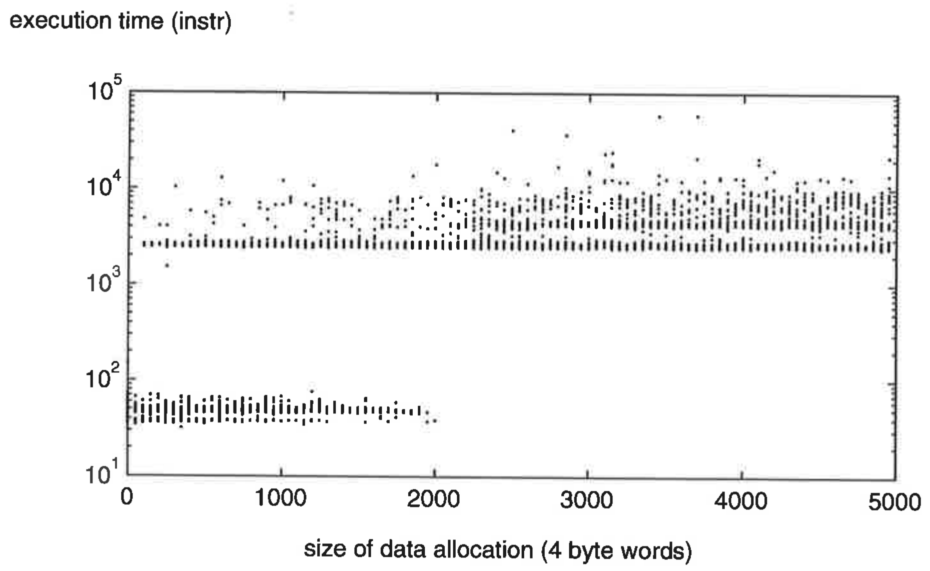


Figure 188. Cost of calls to `malloc`, measured in instructions, on the Sun server platform. 50 calls to `malloc` are made for 100 different block-sizes. The block-size used for each successive call is randomly determined. No deallocation is performed.

performance improves as we allocate larger block sizes. Costs on the PowerPC are more variable than they are on the Sun with some calls on the PowerPC costing more than 100,000 instructions.

Note, that there is a possible bias factor in this experiment: we are allocating the larger blocks from a heap that is more "full" than the heap from which we allocate smaller blocks. We can remove this factor by allocating the large and small blocks in a random order. We show the results of using such a random allocation next.

Allocating large and small blocks in a random order In our second experiment we allocate the same mix of large and small blocks but we permute the sequence of allocations so blocks of different sizes are allocated in some randomly determined order. So, whereas, in the previous experiment, we allocated 50 blocks of one size followed by 50 blocks of the next size up and so on, in this experiment we might allocate a block of 1051 words followed by a block of 251 words followed by a block of one word and so on, in randomly determined order until all 5000 blocks have been allocated.

The results of this experiment for the Sun and PowerPC platforms are shown in figure 188 and 189 respectively.

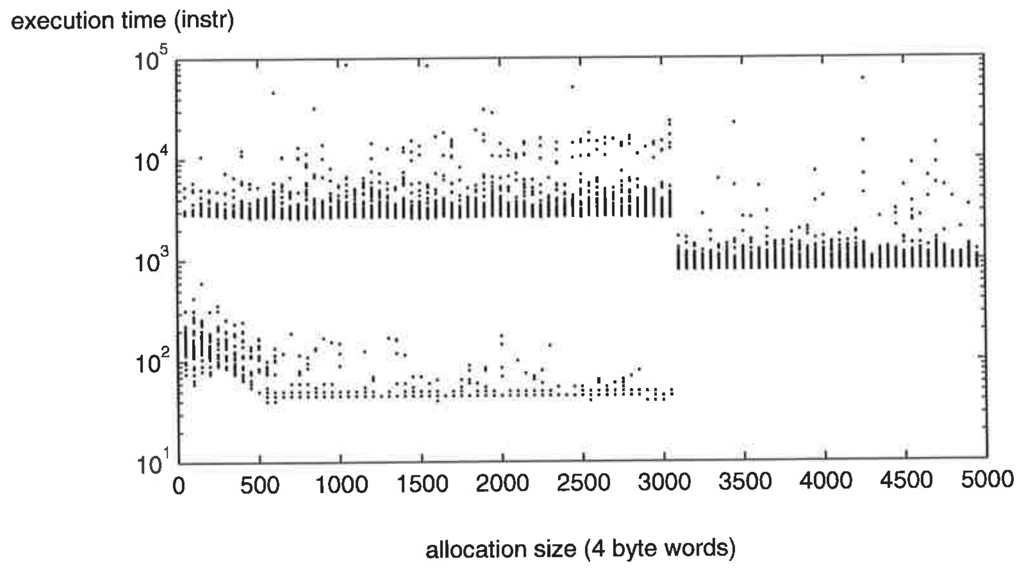


Figure 189. Cost of calls to `malloc`, measured in instructions, on the PowerPC platform. 50 calls to `malloc` are made for 100 different block-sizes. The block-size used for each successive call is randomly determined. No deallocation is performed.

In both figures there are more low-cost allocations of larger blocks and high-cost allocations of smaller blocks. This appears to indicate that at least part of the low cost for smaller blocks in the previous experiment was due to the fact that they were allocated into a less full heap.

However, even with randomised allocation order, smaller blocks typically cost less to allocate than larger blocks. Though, there seems to be no simple function that adequately characterises the cost of `malloc` in these experiments.

Note that there is still scope for improving these experiments. In our implementation, calls to `malloc` will be interspersed with calls to `free`. This on-the-fly freeing up of memory is likely to change the behaviour of `malloc`. We examine the cost of `malloc` when it is interspersed with calls to `free` next.

Cost of `malloc` when immediately followed by `free` We ran experiments to measure the cost of `malloc` when it is immediately followed by a call to `free`. In such circumstances `malloc` is always allocating into a completely empty heap. The results of this experiment for the Sun platform and the PowerPC platform are shown in figure 190 and 191 respectively.

Figure 190 shows a remarkable uniformity of allocation times across the range of block-sizes. Figure 191 displays different behaviour for the lower and upper end of

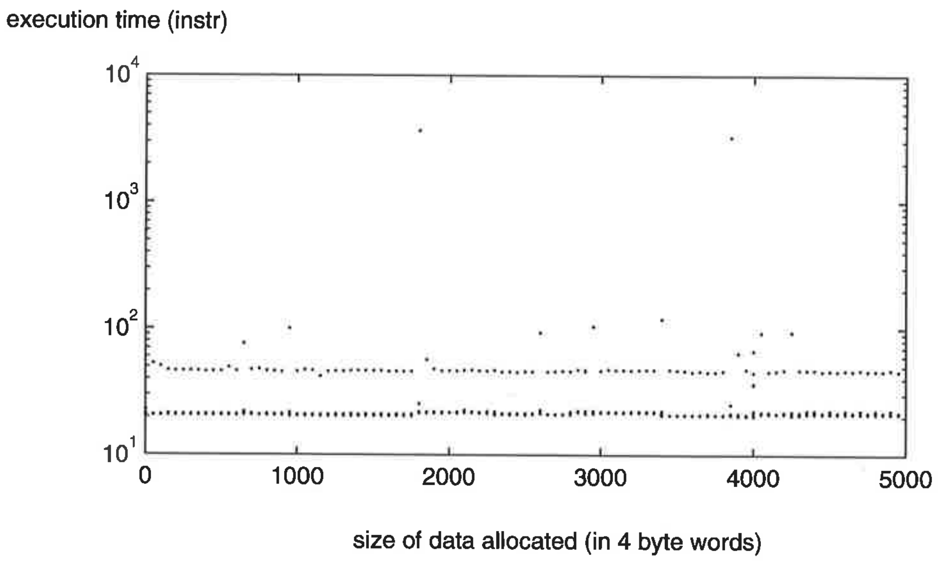


Figure 190. Cost of calls to `malloc`, measured in instructions, on the Sun server platform. 50 calls are made for each block-size and calls are made in ascending order of block-size. Deallocation is performed after each call.

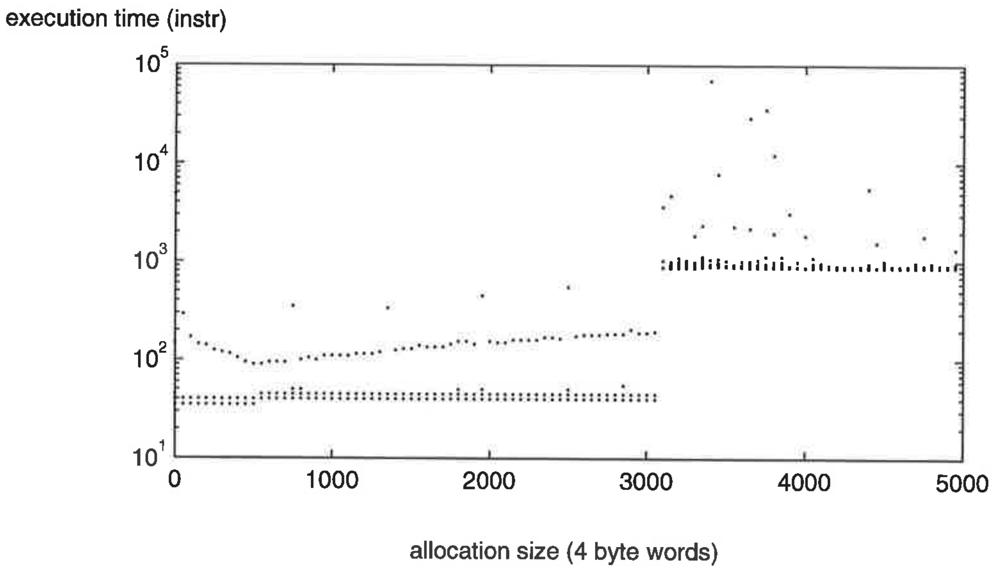


Figure 191. Cost of calls to `malloc`, measured in instructions, on the PowerPC platform. 50 calls are made for each block-size and calls are made in ascending order of block-size.Deallocation is performed after each call.

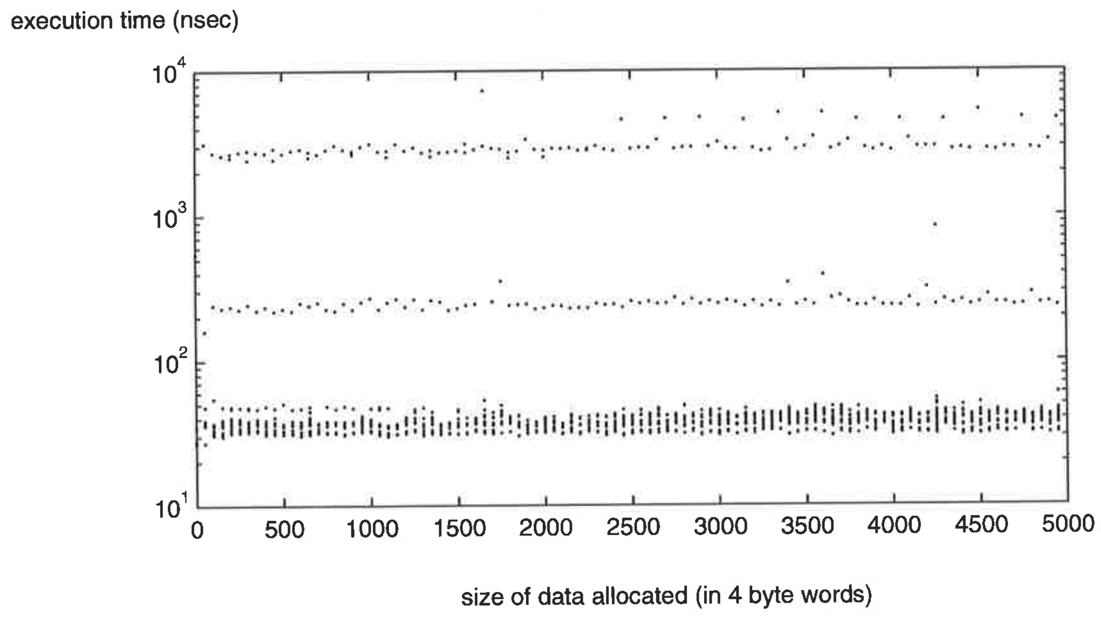


Figure 192. Cost of calls to `malloc`, measured in instructions, on the Sun server platform. 50 calls to `malloc` are made for each block-size followed by 50 calls to `free` these blocks. Calls are made in ascending order of block-size.

the ranges. The sharpness of the discontinuity may indicate a change in algorithm for larger block sizes.

In both cases the variability in allocation costs is low. This uniformity is probably due to the fact that every allocation is into an empty heap.

Note that, like their predecessors, these experiments also fail to accurately capture the conditions likely to prevail in our implementation. The likely scenario in our implementation is to have clusters of allocations, followed by some processing, followed by clusters of deallocation. Our next experiment attempts to mimic this behaviour.

The cost of `malloc` followed by delayed deallocation. In this experiment our programs allocated 50 blocks of data of a given size, reported costs of these allocations, and then freed the 50 blocks of data. The program starts by allocating 50 blocks of one-word and finishes by deallocating 50 blocks of 4951 words. The results of this experiment for the Sun platform and the PowerPC platform are shown in figure 192 and 193 respectively.

In figure 193 as in the previous experiment, there is a very uniform cost for `malloc`'s of different sized blocks of data. Most blocks are allocated within 40 instructions though there are some blocks that take about six times longer than that

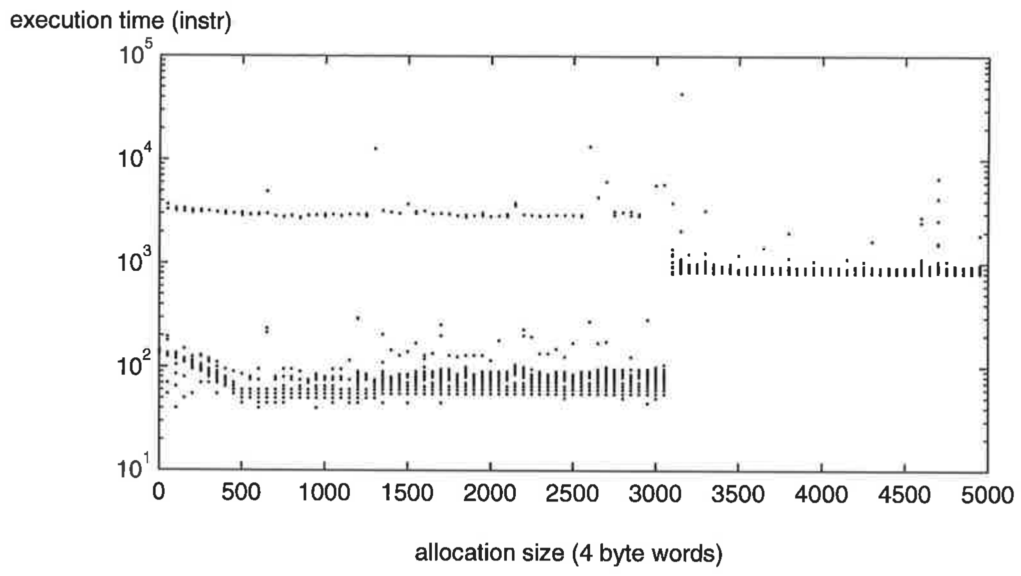


Figure 193. Cost of calls to `malloc`, measured in instructions, on the PowerPC platform. 50 calls to `malloc` are made for each block-size followed by 50 calls to `free` these blocks. Calls are made in ascending order of block-size.

and yet more blocks that take over 50 times longer than the best case to allocate. These outlying allocation times are large enough to significantly affect the average allocation time.

In figure 193 the bulk of measurements lie in the same bands as the previous experiment with the outlying costs for smaller block sizes being higher.

An estimate for the cost of `malloc` Characterisation of the results in these experiments using a cost function or a constant is not easy. A simulator seeking maximum fidelity might seek to model these costs as a set of distributions from which to sample, according to the circumstances under which allocation occurs. However, such a complex model is too heavyweight for our current purposes. Instead we opt to use a constant to characterise the cost of malloc. Our choice of constant must take account of the, mostly, small data sizes used in our simulation experiments. It is best to make the choice of this constant, from the wide range of possible values, conservative so as not to artificially inflate computation to communications ratios in parallel programs. With this in mind we use an estimate of 100 instructions per call. This value is in keeping with the data shown in figures 192 and 193.

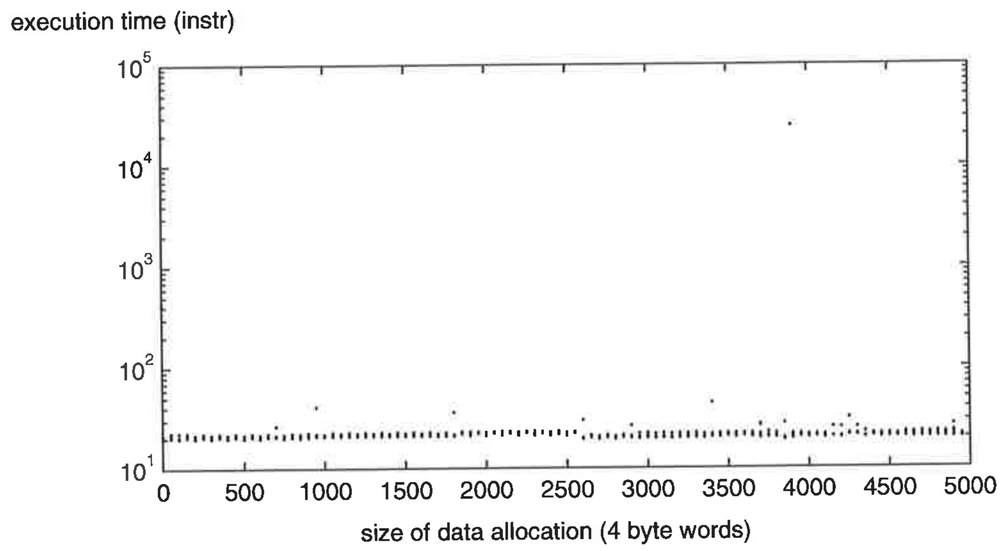


Figure 194. Cost of calls to `free`, measured in instructions, on the Sun server platform. 50 calls are made for each block-size and calls are made in ascending order of block-size. `free` is called immediately after each block is allocated.

F.3.3 The cost of `free`

We measured the cost of `free` in two experiments corresponding to the last two experiments we used to measure the cost of `malloc`. In the first experiment we called `free` immediately after a call to `malloc` which means that the heap contains one block prior to the call. In the second experiment 50 calls to `free` are made after every 50 calls to `malloc`. We present our results next.

Cost of immediate use of `free` This experiment corresponds to the experiment where we measured the cost of `malloc` with immediate deallocation. In this experiment we took away to probes to measure the cost of `malloc` and replaced them with probes to measure the cost of `free`.

Figures 194 and 195 give performance figures for calls to `free` on the Sun and PowerPC architectures respectively.

In both figures there is less variation apparent than the corresponding calls to `malloc` in figures 190 and 191. With the exception of one extreme outlier (at around 3300 words) the cost of `free` on the Sun is approximately 30 instructions. Performance is more varied on the PowerPC and there is a sharp drop in performance for block-sizes 3101 words and over. The cost of `free` for smaller block-sizes is clustered around 70 instructions.

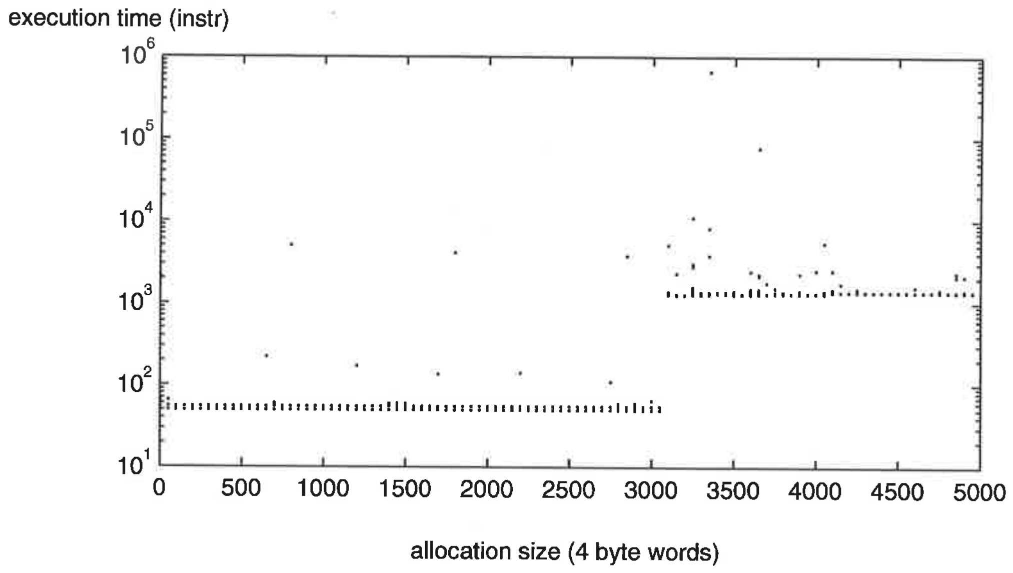


Figure 195. Cost of calls to `free`, measured in instructions, on the PowerPC platform. 50 calls are made for each block-size and calls are made in ascending order of block-size. `free` is called immediately after each block is allocated.

Cost of delayed use of `free` This experiment is a modified version of the experiment that measured the cost of `malloc` with delayed deallocation. We replaced the timing probes surrounding `malloc` with timing probes surrounding `free`.

Figures 196 and 197 show results of these experiments on the Sun and the PowerPC architectures respectively. The scale in figure 196 is linear rather than logarithmic to best capture the range of costs. This figure indicates that `free` is slightly more expensive when calls to `free` are delayed until multiple allocations have been made. A notable exception is the cost of freeing one word of memory. Our timer consistently registered a cost of zero instructions for this activity indicating that the call takes some time less than the actual resolution of the high-resolution clock.

Figure 197 tells a more complex story. For the most part, the cost of deallocation has increased dramatically from the earlier experiment with immediate deallocation. The cost of `free` for most smaller block-sizes is greater than the cost for larger block-sizes[4]. This cost-profile is difficult to characterise with a single scalar or function.

An estimate for `free` A blanket estimate for the cost of `free` is harder to make than for `malloc`. As with `malloc` a simulation of maximum fidelity would employ a

[4]Perhaps this high cost is the reason there seems to be a change of algorithms for larger blocks.

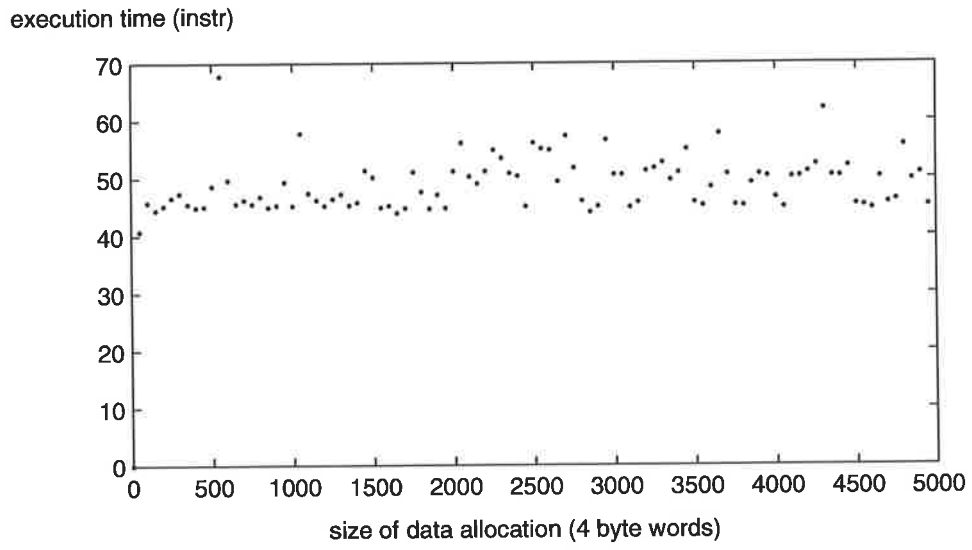


Figure 196. Cost of calls to `free`, measured in instructions, on the Sun server platform. 50 calls are made for each block-size and calls are made in ascending order of block-size. `free` is called after every 50 allocations.

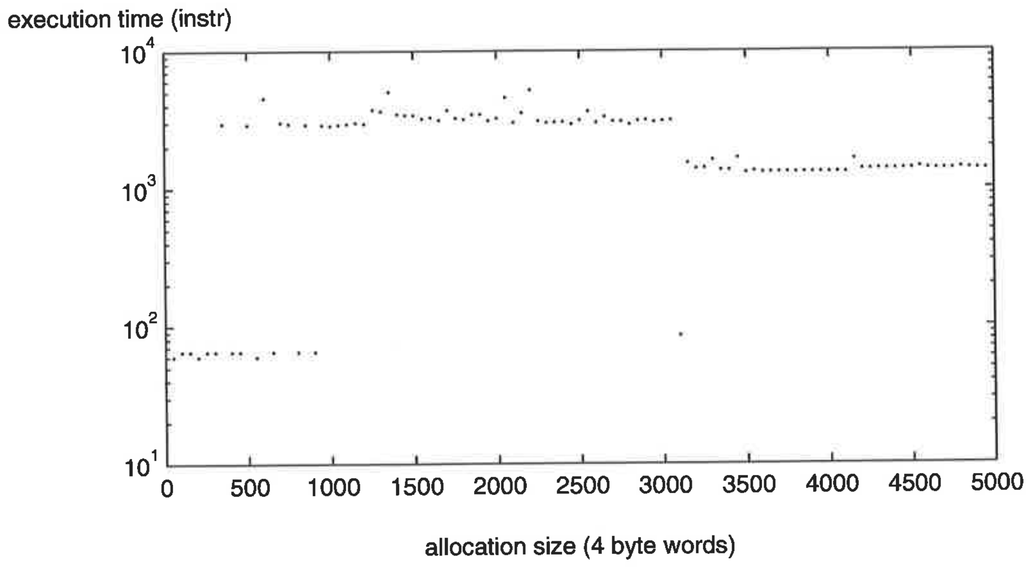


Figure 197. Cost of calls to `free`, measured in instructions, on the PowerPC platform. 50 calls are made for each block-size and calls are made in ascending order of block-size. `free` is called after every 50 allocations.

function to sample from a set of distributions. However, our model only requires a loose approximation of costs. A figure of 70 instructions is a good approximation of the cost of `free` on the Sun. The costs on the PowerPC defy characterisation but a figure of 70 is a reasonable estimate for many smaller block-sizes.

F.4 Conclusion

After sampling the performance of `malloc` and `free` on two architectures on a variety of block-sizes with different patterns of allocation and deallocation we have the following observations:

- The costs of `malloc` and `free` form distributions that depend on the timing of allocation and deallocation, the platform and the size of the data allocated.
- There is no simple function that characterises the cost of `malloc` as a function of block-size. With the exception of a cost discontinuity on the PowerPC platform the cost of `malloc` seems to depend more on how many other blocks are currently allocated in the heap than on the size of the new block being allocated. Even when `malloc` allocates into empty memory performance can vary from call to call.
- There is no simple function that characterises the cost of `free` on both platforms. As with `malloc` the cost of free seems to depend more on the number of blocks already allocated than on the size of the block being freed.
- Cost estimates of 100 instructions for `malloc` and 70 instructions for `free` are not unreasonable as an approximation for a simple model. There is scope to use experimentally derived distributions, such as the ones derived in this chapter, to inform a more detailed model in future.

It must be stressed that any refinement of our cost estimates should involve sampling for a greater variety of block-sizes and using more realistic patterns of allocation and deallocation. The most accurate model for the costs of `malloc` and `free` would probably come from a detailed model of the algorithms used on a particular architecture. The implementation of such a model is beyond the scope of this work.

Bibliography

[1] H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, 1996.

[2] C. Adami, C. Ofria, and T.C. Coller. Evolution of biological complexity. *Proc. Natl. Acad. Sci. USA*, 97:4463–4468, 2002.

[3] M. Aldinucci. Automatic program transformation: The meta tool for skeleton-based languages. In S. Gorlatch and C. Lengauer, editors, *Constructive Methods for Parallel Programming*, Advances in Computation: Theory and Practice, chapter 5. Nova Science Publishers, NY, USA, 2002.

[4] M. Aldinucci, M. Coppola, and M. Danelutto. Rewriting skeleton programs: How to evaluate the data-parallel stream-parallel tradeoff. In S. Gorlatch, editor, *CMPP'98: First International Workshop on Constructive Methods for Parallel Programming*, 1998.

[5] M. Aldinucci, S. Gorlatch, C. Lengauer, and S. Pelagatti. Towards parallel programming by transformation: The FAN skeleton framework. *Parallel Algorithms and Applications*, 16:87–121, 2001.

[6] B. Alexander. Mapping Adl to the Bird-Meertens Formalism. Technical Report 94-18, The University of Adelaide, 1994.

[7] B. Alexander, D. Engelhardt, and A. Wendelborn. A supercomputer implementation of a functional data parallel language. In A.P.W. Böhm and J.T. Feo, editors, *Proceedings of the Conference on High Performance Functional Computing (HPFC'95), Denver, Colorado, USA.* Lawrence Livermore National Laboratory, Livermore, California, USA, 1995.

[8] B. Alexander and A. Wendelborn. Automated transformation of BMF programs. In *The First International Workshop on Object Systems and Software Architectures.*, pages 133–141, 2004.

[9] F.E. Allen and J. Cocke. *A catalogue of optimising transformations, Design and Optimization of Compilers*, pages 1–30. Prentice-Hall, 1971. Editor: R. Rustin.

[10] G. S. Almasi and A. Gottlieb. *Highly parallel computing (2nd ed.)*. Benjamin-Cummings Publishing Co., Inc., 1994.

[11] E. A. Ashcroft, A. A. Faustini, and R. Jagannathan. An intensional language for parallel applications programming. In B.K.Szymanski, editor, *Parallel Functional Languages and Compilers*, pages 11–49. ACM Press, 1991.

[12] B. Bacci, S. Gorlatch, C. Lengauer, and S. Pelagatti. Skeletons and transformations in an integrated parallel programming environment. In *Parallel Computing Technologies (PaCT-99)*, LNCS 1662, pages 13–27. Springer-Verlag, 1999.

[13] J. Backus. Can programming be liberated from the von Neumann style? a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613 – 641, August 1978.

[14] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Survey*, 26(4), December 1999.

[15] C. R. Banger and D. B. Skillicorn. A foundation for theories of arrays. Technical report, January 1993. Obtainable at `ftp.qucis.queensu.ca:/pub/skill/`.

[16] R. Bernecky. Apex - the APL parallel executor. Master's thesis, University of Toronto, 1997.

[17] R. Bird. A calculus of functions for program derivation. Technical Report 64, Programming Research Group, Oxford University, 1987.

[18] R. Bird. Lectures in constructive functional programming. In *Constructive Methods in Computing Science*, pages 151 – 216. NATO ASI Series, Vol F55, 1989.

[19] R. Bird. *Introduction to Functional Programming Using Haskell.* Prentice Hall, 1998. Second Edition.

[20] R. Bird and O. de Moor. *The Algebra of Programming.* International Series in Computer Science. Prentice-Hall, 1996.

[21] R S Bird. Algebraic identities for program calculation. *The Computer Journal*, 32(2):122–126, 1989.

[22] G. E. Blelloch. *Vector Models for Data-Parallel Computing.* The MIT press, Cambridge Massachusetts, 1990.

[23] Guy E. Blelloch, Jonathan C. Hardwick, Jay Sipelstien, Marco Zahga, and Siddhartha Chatterjee. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing*, 21(1):4–14, April 1994.

[24] P. Borras, D. Clément, Th. Despeyroux, J. Incerpi, G. Kahn, et al. CENTAUR: the system. *SIGSOFT software engineering notes / SIGPLAN: SIGPLAN Notices*, 24(2), February 1989.

[25] T. A. Bratvold. A Skeleton-Based Parallelising Compiler for ML. In *Proceedings of the Fifth International Workshop on Implementation of Functional Languages, Nijmegen, the Netherlands*, pages 23–34, September 1993.

[26] D. Burger, S. W. Keckler, K. S. McKinley, M. Dahlin, Lizy K. J, C Lin, C. R. Moore, J. Burrill, R. G. McDonald, W. Yoder, and the TRIPS Team. Scaling to the end of silicon with edge architectures. *Computer*, 37(7):44–55, 2004.

[27] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *J. ACM*, 24(1):44–67, 1977.

[28] B. Carpentieri and G. Mou. Compile-time transformations and optimization of parallel divide-and-conquer algorithms. *ACM SIGPLAN Notices*, 26(10):19–28, October 1991.

[29] M. M. T. Chakravarty and G. Keller. More types for nested data parallel programming. In Philip Wadler, editor, *Proceedings of the Fifth ACM*

SIGPLAN International Conference on Functional Programming (ICFP'00), pages 94–105. ACM Press, 2000.

[30] M. M. T. Chakravarty, G. Keller, R. Lechtchinsky, and W. Pfannenstiel. Nepal - nested data parallelism in Haskell. In *Proceedings of the 7th International Euro-Par Conference Manchester on Parallel Processing*, pages 524–534. Springer-Verlag, 2001.

[31] S. Ciarpaglini, L. Folchi, and S. Pelagatti. Anacleto: User manual. Technical report, Dipartimento di Informatica, Universit di Pisa., 1998.

[32] C. Click and K. D. Cooper. Combining analyses, combining optimizations. *ACM Transactions on Programming Languages and Systems*, 17(2):181–196, March 1995.

[33] W F. Clocksin and C. S. Mellish. *Programming in Prolog (2nd ed.)*. Springer-Verlag New York, Inc., 1984.

[34] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, 1989.

[35] M. Cole. List homomorphic parallel algorithms for bracket matching. Technical Report CSR-29-93, The University of Edinburgh, Author's email address: `mic@dcs.ed.ac.uk`, 1993.

[36] M. Cole. Parallel programming, list homomorphisms and the maximum segment sum problem. Technical Report CSR-25-93, The University of Edinburgh, Author's email address: `mic@dcs.ed.ac.uk`, 1993.

[37] D. C. Crooke. *Practical Structured Parallelism Using BMF*. PhD thesis, University of Edinburgh, 1999.

[38] A. Cunha and S. P. Pinto. Making the point-free calculus less pointless. In *Proceedings of the 2nd APPSEM II Workshop*, pages 178–179, April 2004. Abstract.

[39] A. Cunha, S. P. Pinto, and J. Proenca. Down with variables. Technical Report DI-PURe-05.06.01, June 2005.

[40] M. A. Cunha. Point-free programming with hylomorphisms. In *Workshop on Datatype-Generic Programming, Oxford*, June 2004.

[41] M. Danelutto, F. Pasqualetti, and S. Pelagatti. Skeletons for Data Parallelism in P3L. In C. Lengauer, M. Griebl, and S. Gorlatch, editors, *Proceedings of the 3rd European Conference on Parallel Processing (Euro-Par'97), Passau, Germany*, volume 1300 of *Lecture Notes in Computer Science*, pages 619–628. Springer-Verlag, Berlin, Germany, 1997.

[42] 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'93: Parallel Architectures and Languages Europe*. Springer-Verlag, June 1993. LNCS 694.

[43] J. Darlington, Y. Guo, H. W. To, and J. Yang. Functional skeletons for parallel coordination. In *Proceedings of the First International Euro-Par Conference on Parallel Processing*, pages 55–66. Springer-Verlag, 1995.

[44] J. Darlington, Y. K. Guo, H. W. To, and J. Yang. Parallel skeletons for structured composition. In *Proceedings of the fifth ACM SIGPLAN symposium on Principles & practice of parallel programming*, pages 19–28. ACM Press, 1995.

[45] A. Darte. Mathematical tools for loop transformations: from systems of uniform recurrence equations to the polytope model. Technical Report RR97-26, Ecole Normale Suprieure de Lyon, 1997.

[46] M. de Jonge and J. Visser. Grammars as contracts. *Lecture Notes in Computer Science*, 2177, 2001.

[47] O. de Moor and G. Sittampalam. Generic program transformation. In *Advanced Functional Programming*, pages 116–149, 1998.

[48] S. J. Deitz, B. L. Chamberlain, S. E. Choi, and L. Snyder. The design and implementation of a parallel array operator for the arbitrary remapping of data. *SIGPLAN Not.*, 38(10):155–166, 2003.

[49] B. Desai. *An Introduction to Database Systems*. West Publishing, 1991.

[50] C. Ding and K. Kennedy. The memory bandwidth bottleneck and its amelioration by a compiler. In *Int. Parallel and Distributed Processing Symposium*, pages 181–190, 2000.

[51] J. Dongarra et al. *A Users' Guide to PVM Parallel Virtual Machine.* Oak Ridge National Laboratory, Knoxville, TN, USA, July 1991.

[52] C. Elliott, S. Finne, and O. de Moor. Compiling embedded languages. In *SAIG*, pages 9–27, 2000.

[53] D. Engelhardt. *A Generalised Execution Model for Data-Parallel Computing.* PhD thesis, School of Computer Science, University of Adelaide, 1997.

[54] D.C. Engelhardt and A.L. Wendelborn. A partitioning-independent paradigm for nested data parallelism. *International Journal of Parallel Programming*, 24(4):291–317, August 1996.

[55] J T Feo and D C Cann. A report on the SISAL language project. *Journal of Parallel and Distributed Computing*, pages 349–366, December 1990.

[56] S.M. Fitzgerald and R.R. Oldehoeft. Update-in-place Analysis for True Multidimensional Arrays. In A.P.W. Böhm and J.T. Feo, editors, *Proceedings of the Conference on High Performance Functional Computing (HPFC'95), Denver, Colorado, USA*, pages 105–118. Lawrence Livermore National Laboratory, Livermore, California, USA, 1995.

[57] I. Foster. *Designing and building parallel programs:Concepts and tools for parallel software.* Addison-Wesley, 1995.

[58] J. Gibbons. An introduction to the Bird-Meertens formalism. In *New Zealand Formal Program Development Colloquium Seminar, Hamilton*, November 1994.

[59] J. Gibbons. A pointless derivation of radix sort. *J. Funct. Program.*, 9(3):339–346, 1999.

[60] J. Gibbons and G. Jones. The under-appreciated unfold. In *Proceedings 3rd ACM SIGPLAN Int. Conf. on Functional Programming, ICFP'98, Baltimore, MD, USA, 26–29 Sept. 1998*, volume 34(1), pages 273–279. ACM Press, New York, 1998.

[61] S. Gorlatch. Stages and transformations in parallel programming. In *Abstract Machine Models*, pages 147–161. IOS Press, 1996.

[62] S. Gorlatch. Optimizing compositions of scans and reductions in parallel program derivation. Technical Report MIP-9711, Fakultät für Mathematik und Informatik, Universität Passau, 1997.

[63] S. Gorlatch. Abstraction and performance in the design of parallel programs. Technical Report MIP-9802, Universität Passau, January 1998.

[64] S. Gorlatch and C. Lengauer. Abstraction and performance in the design of parallel programs: An overview of the SAT approach. *Acta Informatica*, 36(9–10):761–803, May 2000.

[65] B. Greer, J. Harrison, G. Henry, W. Li, and P. Tang. Scientific computing on the Itanium processor. In *Proceedings of the 2001 Conference on Supercomputing*, 2001. Available on the Web as `http://www.sc2001.org/papers/pap.pap266.pdf`.

[66] D. Grove and P. Coddington. Precise MPI performance measurement using MPIBench. In *In Proceedings of HPC Asia*, September 2001.

[67] M. F. Guest. Communications benchmarks on high-end and commodity-type computers, April 2002. CCLRC Daresbury Laboratory, URL: www.dl.ac.uk/CFS/benchmarks/pmb.2003/pmb.2003.pdf.

[68] M. Gupta, E. Schonberg, and H. Srinivasan. A unified framework for optimizing communication in data-parallel programs:. *IEEE Transactions on Parallel and Distributed Systems*, 7(7):689–704, 1996.

[69] M. Haines and W. Bohm. Task management, virtual shared memory, and multithreading in a distributed memory implementation of SISAL. In *Proc. of Parallel Architectures and Languages Europe (PARLE'93), LNCS 694*, pages 12–23. Springer-Verlag, june 1993.

[70] G. Hains and L. M. R. Mullin. Parallel functional programming with arrays. *The Computer Journal*, 36(3):238 – 245, 1993.

[71] M. Hamdan. *A Combinational Framework for Parallel Programming Using Algorithmic Skeletons.* PhD thesis, Department of Computing and Electrical Engineering. Heriot-Watt University, 2000.

[72] J. Hammes and W. Böhm. *Research Directions in Parallel Functional Programming, K. Hammond and G.J. Michaelson (eds)*, chapter Memory Performance of Dataflow Programs. Springer-Verlag, 1999.

[73] M. T. Heath and J. A. Etheridge. ParaGraph: a tool for visualizing performance of parallel programs. URL: http://www.csar.uiuc.edu/software/paragraph/, 1992.

[74] M. Hennessy. *The Semantics of Programming Languages.* Wiley, 1990.

[75] J. M. D. Hill, B. McColl, D. C. Stefanescu, M. W. Goudreau, K. Lang, S. B. Rao, T. Suel, T. Tsantilas, and R. H. Bisseling. BSPlib: The BSP programming library. *Parallel Computing*, 24(14):1947–1980, 1998.

[76] J. M.D. Hill and D. B. Skillicorn. Lessons learned from implementing BSP. *Future Gener. Comput. Syst.*, 13(4-5):327–335, 1998.

[77] Jonathan M. D. Hill, Keith M. Clarke, and Richard Bornat. Parallelizing imperative functional programs: the vectorization monad. *J. Symb. Comput.*, 21(4-6):561–576, 1996.

[78] Z. Hu. Optimization of skeletal parallel programs (invited talk ohp slides). In *3rd International Workshop on Constructive Methods for Parallel Programming (CMPP 2002)*, 2002.

[79] Z. Hu, H. Iwasaki, and M. Takeichi. An accumulative parallel skeleton for all. In *European Symposium on Programming*, pages 83–97, 2002.

[80] Zhenjiang Hu, Hideya Iwasaki, Masato Takeichi, and Akihiko Takano. Tupling calculation eliminates multiple data traversals. In *Proceedings 2nd ACM SIGPLAN Int. Conf. on Functional Programming, ICFP'97, Amsterdam, The Netherlands, 9–11 June 1997*, volume 32(8), pages 164–175. ACM Press, New York, 1996.

[81] P. Hudak, J. Peterson, and J. Fasel. A gentle introduction to Haskell 98. `http://www.haskell.org/tutorial/`, 1999.

[82] Paul Hudak and Jonathan Young. A collecting interpretation of expressions (without powerdomains). In *ACM Principles of Programming Languages*, pages 107–118, 1988.

[83] Myricom Inc. Myricom home page:. URL: http://www.myri.com/.

[84] C. B. Jay. A semantics for shape. *Science of Computer Programming*, 25(2–3):251–283, 1995.

[85] C. B. Jay. The FISh language definition. Technical report, School of Computer Science, University of Technology Sydney, 1998.

[86] C. B. Jay. *Research Directions in Parallel Functional Programming, K. Hammond and G.J. Michaelson (eds)*, chapter 9: Shaping Distributions, pages 219–232. Springer-Verlag, 1999.

[87] G. Jones and M. Sheeran. The study of butterflies. In *Collecting butterflies (Technical Monograph PRG-91)*. Oxford University Computing Laboratory, 1991.

[88] Mark P. Jones. Hugs 1.3, the Haskell User's Gofer System: User manual. Technical Report NOTTCS-TR-96-2, Functional Programming Research Group, Department of Computer Science, University of Nottingham, August 1996.

[89] G. Kahn. Natural semantics. In *Fourth Annual Symposium on Theoretical Aspects of Computer Science*, volume 247 of *LNCS*. Springer-Verlag, 1987.

[90] L. V. Kalé, S. K., and K. Varadarajan. A framework for collective personalized communication. In *International Parallel and Distributed Processing Symposium (IPDPS'03)*, April 2003.

[91] G. Keller and M. M. T. Chakravarty. Flattening trees. In *European Conference on Parallel Processing*, pages 709–719, 1998.

[92] G. Keller and N. Simons. A calculational approach to flattening nested data parallelism in functional languages. In J. Jaffar and R. H. C. Yap, editors, *Concurrency and Parallelism, Programming, Networking, and Security: Second Asian Computing Science Conference, ASIAN'96*, volume 1179 of *Lecture Notes in Computer Science*, pages 234–243. Springer-Verlag, 1996.

[93] K. Kennedy. *A Survey of Data Flow Analysis Techniques*, pages 5–54. Prentice-Hall, 1981.

[94] T. Kisuki, P. M. W. Knijnenburg, M. F. P. O'Boyle, F. Bodin, and H. A. G. Wijshoff. A feasibility study in iterative compilation. In *ISHPC*, pages 121–132, 1999.

[95] H. Kuchen. A skeleton library. In *Euro-Par Conference on Parallel Processing*. Springer-Verlag, 2002.

[96] Quadrics Ltd. Quadrics home page:. http://www.quadrics.com/.

[97] S. Mahlke. Design and implementation of a portable global code optimizer, 1991.

[98] U. Martin and T. Nipkow. Automating Squiggol. In M. Broy and C.B. Jones, editors, *Programming Concepts and Methods*, pages 233–247. North-Holland, 1990.

[99] P. Martinaitis. Simulation and visualisation of parallel BMF code. Honours Thesis, School of Computer Science, University of Adelaide, 1998.

[100] M. Mauny and A. Suárez. Implementing functional languages in the categorical abstract machine. In *LFP '86: Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 266–278, New York, NY, USA, 1986. ACM Press.

[101] Sun Microsystems. Sun mpi 5.0 programming and reference guide. URL: http://www.sun.com/products-n-solutions/hardware/docs/pdf/816-0651-10.pdf.

[102] Sun Microsystems. The Java HotSpot Virtual Machine Technical white paper, 2001. Available at 'http://java.sun.com/products/hotspot/index.html'.

[103] R. Miller. *A constructive theory of Multidimensional Arrays.* PhD thesis, Lady Margret Hall, Oxford, February 1993.

[104] M. Mottl. Automating functional program transformation. Master's thesis, Division of Informatics, University of Edinburgh, September 2000.

[105] Message Passing Interface Forum MPIF. MPI-2: Extensions to the Message-Passing Interface. Technical Report, University of Tennessee, Knoxville, 1996.

[106] DPCE Subcommittee Numerical C Extensions Group of X3J11. Data Parallel C Extensions. Technical Report X3J11/94-080, X3J11, 1994.

[107] University of Washington CSE Department. ZPL homepage. http://www.cs.washington.edu/research/zpl/home/index.html.

[108] Y. Onoue, Z. Hu, M. Takeichi, and H. Iwasaki. A calculational fusion system HYLO. In *Proceedings of the IFIP TC 2 WG 2.1 international workshop on Algorithmic languages and calculi*, pages 76–106. Chapman & Hall, Ltd., 1997.

[109] P. Pepper, J. Exner, and M. Südholt. Functional development of massively parallel programs. In *Formal methods in programming and their applications, Lecture Notes in Computer Science, Volume 735*, June 1993.

[110] S. L. Peyton-Jones and E. Meijer. Henk: a typed intermediate language. In *Proceedings of the Types in Compilation Workshop*, Amsterdam, June 1997.

[111] D. Philp. Mapping parallel BMF constructs to a parallel machine. Master's thesis, School of Computer Science, University of Adelaide, 2003.

[112] J. Prins and D. Palmer. Transforming high-level data-parallel programs into vector operations. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 119–128, San Diego, CA., May 19-22, 1993. ACM.

[113] Paul H. J. Kelly Q. Wu, A. J. Field. M-tree: A parallel abstract data type for block-irregular adaptive applictions. In *Euro-Par '97 Parallel Processing, Third International Euro-Par Conference, Passau, Germany, August 26-29, 1997, Proceedings*, volume 1300 of *Lecture Notes in Computer Science*, pages 638–649. Springer-Verlag, 1997.

[114] Backhouse R. An exploration of the Bird-Meertens formalism. Technical Report CS-8810, Groningen University, 1988.

[115] K. P. Nikhil R. S. Arvind. I-structures: Data structures for parallel computing. *ACM Trans. Program. Lang. Syst.*, 11(4):598–632, 1989.

[116] S. Rajopadhye. LACS: a language for affine communication structures. Technical Report RR-2093, INRIA Rennes, 1993.

[117] P. Rao and C. Walinsky. An equational language for data-parallelism. In *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 112–118. ACM Press, 1993.

[118] H. Richardson. High performance fortran: history, overview and current developments. Technical report, 1996.

[119] P. Roe. Adl: The adelaide language. Technical report, University of Adelaide, July 1992.

[120] P. Roe. Derivation of efficient data parallel programs. In *17th Australasian Computer Science Conference*, pages 621 – 628, January 1994.

[121] J. Roorda. Pure type systems for functional programming. Master's thesis, University of Utrecht, 2000. INF/SCR-00-13.

[122] U. Rüde. Iterative algorithms in high performance architectures. In *Euro-Par'97, Parallel Processing, Third International Euro-Par Conference, Passau, Germany, August 26-29, 1997*, pages 57–71. Springer, 1997.

[123] D. Sands. Total correctness by local improvement in the transformation of functional programs. *ACM Trans. Program. Lang. Syst.*, 18(2):175–234, 1996.

[124] P. Sheridan. The arithmetic translator compiler of the IBM FORTRAN Automatic Coding System. *Communications of the ACM*, 2(2):9–21, 1959.

[125] D. Skillicorn. *Foundations of parallel programming.* Cambridge University Press, 1995.

[126] D. B. Skillicorn. Architecture independent parallel computation. *IEEE Computer*, pages 38 – 49, December 1990.

[127] D. B. Skillicorn. Costs and implementations of bulk data types. In *Bulk Data Types for Architecture Independence, British Computer Society, Parallel Processing Specialist Group*, 1994.

[128] D. B. Skillicorn and W. Cai. Equational code generation: Implementing categorical data types for data parallelism. In *TENCON '94, Singapore.* IEEE, 1994.

[129] D. B. Skillicorn, M. Danelutton, S. Pelagatti, and A. Zavanella. Optimising Data-Parallel Programs Using the BSP Cost Model. In *EuroPar'98*, volume 1470, pages 698–708. Springer-Verlag, 1998.

[130] David B. Skillicorn and Domenico Talia. Models and languages for parallel computation. *ACM Computing Surveys*, 30(2):123–169, 1998.

[131] M. Spivey. A categorical approach to the theory of lists. In *Mathematics of Program Construction*, pages 400–408. Springer-Verlag, June 1989. Found in: Lecture Notes in Computer Science Vol 375.

[132] G. L. Steele, Jr. and W. D. Hillis. Connection machine lisp: fine-grained parallel symbolic processing. In *Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 279–297. ACM Press, 1986.

[133] S. Swanson, K. Michelson, A. Shwerin, and M. Oskin. Dataflow: The road less complex. In *The Workshop on Complexity-effective Design (WCED),held in conjunction with the 30th Annual International Symposium on Computer Architecture (ISCA)*, June 2003.

[134] Thinking Machines Corporation, Cambridge, MA. *CM Fortran Reference Manual*, 2.1 edition, 1994.

[135] P.W. Trinder, H-W. Loidl, E. Barry Jr., K. Hammond, U. Klusik, S.L. Peyton Jones, and Á.J. Rebón Portillo. The Multi-Architecture Performance of the Parallel Functional Language GPH. In A. Bode, T. Ludwig, and R. Wismüller, editors, *Euro-Par 2000 — Parallel Processing*, LNCS, Munich, Germany, 29.8.-1.9., 2000. Springer-Verlag.

[136] M. Tullsen. The Zip Calculus. In Roland Backhouse and Jose Nuno Oliveira, editors, *Mathematics of Program Construction, 5th International Conference,*

MPC 2000, volume 1837 of *Lecture Notes in Computer Science*, pages 28–44. Springer-Verlag, July 2000.

[137] L. G. Valiant. A Bridging Model for Computation. *Communications of the ACM*, 33(8):103–111, 1990.

[138] E. Visser. Strategic pattern matching. In *RtA '99: Proceedings of the 10th International Conference on Rewriting Techniques and Applications*, pages 30–44. Springer-Verlag, 1999.

[139] E. Visser. Stratego: A language for program transformation based on rewriting strategies. In *RTA '01: Proceedings of the 12th International Conference on Rewriting Techniques and Applications*, pages 357–362. Springer-Verlag, 2001.

[140] E. Visser. A survey of strategies in program transformation systems. *Electronic Notes in Theoretical Computer Science*, 57, 2001.

[141] R. von Hanxleden, K. Kennedy, C. H. Koelbel, R. Das, and J. H. Saltz. Compiler analysis for irregular problems in Fortran D. In *1992 Workshop on Languages and Compilers for Parallel Computing*, number 757 in Lecture Notes in Computer Science, pages 97–111, New Haven, Conn., 1992. Berlin: Springer Verlag.

[142] P. Wadler. The concatenate vanishes. Technical report, Department of Comp. Sci., University of Glasgow., 1988.

[143] P. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231–248, 1990.

[144] C. Walinsky and D. Banerjee. A functional programming language compiler for massively parallel computers. In *Proceedings of the 1990 ACM conference on LISP and functional programming*, pages 131–138. ACM Press, 1990.

[145] C. Walinsky and D. Banerjee. A Data-Parallel FP Compiler. *Journal of Parallel and Distributed Computing*, 22:138–153, 1994.

[146] J. Windows. Automated parallelisation of code written in the Bird-Meertens Formalism. Honours Thesis, School of Computer Science, University of Adelaide, 2003.