University of London
Imperial College of Science Technology and Medicine
Department of Computing

Exploiting Data-Parallelism in Functional Languages

Guido Karel Jouret

Submitted in part fulfillment of the requirements for the degree of Doctor of Philosophy in Engineering of the University of London and the Diploma of Imperial College of Science Technology and Medicine, Friday, September 27, 1991

ABSTRACT

Existing sequential languages are inherently unsuitable for data-parallel programming because of the von-Neumann execution model built-in to the semantics of these languages. Consideration of the requirements for data-parallel programming leads to the development of a declarative language based on the functional style.

The powerful abstraction mechanism provided by functional languages allow the capabilities of data-parallel architectures to be presented via a set of higher-order primitive operations defined on an underlying array data-type. Functional programs consisting of composition of these data-parallel operations therefore naturally exhibit data-parallelism.

A set of appropriate primitive data-parallel operations is added to a simple functional language. Derived operations are defined in terms of this initial set and sample application programs demonstrate the elegance and advantages of data-parallel programming in a functional language.

A compilation scheme for compiling this extended functional language to an abstract data-parallel machine architecture is developed. An abstract data-parallel architecture, the *Planar Abstract Machine* (PAM) is presented. PAM is a dual-instruction architecture as two forms of basic instructions exist: those that operate on *scalar* (word-size) values, and those that take *planar* (multiple word) values as operands. A compilation scheme generates planar-code from conventional, user-defined functions, translating all basic operations and control structures into planar forms suitable for data-parallel execution. This permits the use of fully-general user-defined functions (e.g. with recursion, conditional statements, algebraic data-types) in parallel.

Efficiency concerns and possible optimizations are discussed. The development of a methodology for optimizing and parallelizing programs by the transformation of computation and communication operations is developed. Further possible avenues of research, including the development of primitive operations on alternative aggregate data-structures, implementation of the data-parallel model of computation on MIMD systems, and the role of transformation in the possible development of systolic algorithms are discussed.

He who sees the Infinite in all things sees God.
He who sees the Ratio only sees himself only.
Therefore God becomes as we are, that we may be as he is.

– *William Blake*

ACKNOWLEDGEMENTS

A thesis is an individual and often solitary journey on the path to understanding and knowledge. The satisfaction gained at the conclusion is not from arriving, but from what has been learned along the way, both about the subject matter at hand and of one's potentials and limitations. A number of individuals have been instrumental in my education and deserve particular acknowledgement.

My supervisor John Darlington provided unfailing support, freedom, and encouragement, consistent since the beginning, when I hadn't the foggiest idea of what I was talking about. Advice (both solicited and unsolicited) was generously provided by David Lillie, Andrew Bennett, Ross Paterson, David Sharp, and many of the other members of the Functional Programming Section at Imperial. Tony Field and Peter Harrison elicit my gratitude for their forthright (constructive) criticisms which improved the presentation of the thesis considerably.

I never would have considered trying for a Ph.D. if it weren't for the moral-support provided by my world-wide circle of friends, specifically: Hilary Wilkinson, Chris Marriott, Alan Spidle, Steve Fernandez, and most of all, Kenn Frankel, whose confidence in my abilities vastly exceeded my own. My brother Dirk and my sister Erika have supported me since the beginning. For putting up with my anxieties and worries over these past few years, and for never losing faith in me, Nadège Ferrero gets my deepest thanks.

I dedicate this thesis to my parents, who never let me think that I wasn't up to the task and who selflessly invested a great deal of time, love, and encouragement to make this thesis happen. Over a period of 25 years we moved across 3 continents, 6 schools, and 4 languages but throughout all these changes, their commitment and devotion remained undiminished. This thesis is for them.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

1. Introduction

The data-parallel model of computation exhibits great promise but is an underdeveloped area of research. Language design for data-parallel architectures has been largely restricted to adapting sequential imperative languages, an approach that is inherently flawed because of an assumed von-Neumann execution model underlying these languages.

Recognizing the deficiencies in sequential imperative languages for data-parallel programming leads to the identification of the features required for the exploitation of data-parallelism. These include: aggregate data-types supported by higher-order monolithic operations, no globally-accessible state, strict enforcement of locality, constructs to express communication, and freedom from side-effects. These requirements suggest the development of a simple declarative language; extended with data-parallel operations and consistent with the functional style.

1.1. Motivation

The data-parallel model of computation promises massive scalable parallelism, inherent load-balancing, and low overhead. Nevertheless, in the last decade, the process-parallel model of computation has been the main avenue to parallelism. Part of the success of the process-parallel model stems from the fact that languages for sequential von-Neumann architectures can be adapted to exploit process-parallelism by adding constructs to support inter-process communication. Adapting languages developed for sequential von-Neumann machines to parallel architectures makes the underlying architectural assumptions built in to the semantics of these languages readily apparent. Nowhere are these underlying assumptions more limiting than in the extension of sequential imperative languages for programming data-parallel machines.

Existing sequential imperative languages are unsuitable for data-parallel programming. Nevertheless, some of these languages, most notably Fortran, have been adapted for data-parallel programming but the sequential origins of these languages make data-parallel programming unnecessarily difficult. Languages for data-parallelism have to be liberated from the von-Neumann constraints of sequential languages.

1.1.1. Exploiting Parallelism in the Presence of Dependencies

Parallelism can be realized whenever independent activity is permitted. Dependencies on data and control constrain parallelism. Parallel execution of programs written in sequential imperative languages is hampered by the lack of any explicit constructs to denote parallel computation. In the absence of explicit constructs a compiler must attempt to ferret out parallelism by analyzing programs. The automatic extraction of parallelism is complicated by the need to respect an underlying sequential operational semantics.

Imperative languages permit destructive assignment to variables which imposes additional dependencies. There are three types of dependencies on data:

- *flow-dependencies*: a variable may not be used before it has been assigned a value.

- *anti-dependencies*: the content of a variable must be used by all of its consumers before it can be overwritten.

- *output-dependencies*: multiple-assignments to the same variable must be performed in the order specified by sequential execution of the program.

The only "true" dependency is flow-dependency. The other forms of dependencies are a consequence of the use of destructive assignment and only serve to introduce additional constraints on parallelism.

1.1.2. von-Neumann Constraints on Imperative Languages

The semantics of most sequential imperative languages reflect the following architectural assumptions:

- The availability of a global state, accessible by any portion of the program.
- The uniformity of the state (i.e. all locations in the store are functionally equivalent).
- The existence of a single *mutator* (i.e. only one agent is manipulating the state with the consequence that all locations not currently accessed by the mutator remain unchanged).

Massively parallel machines rely on distributing the global store into distinct local stores, each accessed by a local processor. Sequential imperative languages assume that the state is globally-accessible and completely uniform (i.e. all areas of memory incur the same access cost). Fortran, for example, permits any part of the global state to be referenced in any statement, as in:

```
A(i) = B(j)
```

where `i` and `j` may be any indices in the range of the arrays `A` and `B`, which fails to take into account where elements of `A` and `B` may be allocated. Access to any variable in the state is permitted. Distribution of the machine state in parallel systems ensures that access to elements in the store is no longer uniform. Non-local access requires explicit communication with resulting latency due to network contention and bandwidth restrictions. If contention-free inter-processor communication facilities are unavailable communication requests are serialized unnecessarily. Contention-free communication is a product of communication patterns and network topology. Exploiting parallelism in communication becomes important. Languages such as Fortran do not enforce any notion of locality because the access-mechanism of variables is overly general: the programmer is not made aware of the cost of accessing non-local variables.

In any parallel machine there are multiple mutators and the dependencies in the program (when they can be detected) constrain the independent activity of the mutators. When dependencies cannot be detected costly synchronization or unnecessary sequential

execution is required to reach a known common machine state prior to indulging in further parallel activity.

1.1.3. Data-Parallel Programming

Arrays, bags, sets, queues, lists, trees, and graphs are all examples of commonly used aggregates. Data-parallel programming relies on the availability of aggregate data-structures and constructs in the language which exhibit data-parallelism in computation and communication on those aggregates.

The uniformity of the store in sequential von-Neumann machine architectures has biased language design towards aggregate data-structures which are efficient on those architectures. The array is the most commonly used aggregate because indices correspond directly to addresses in the store and dedicated hardware support is provided for quick access to array elements (e.g. auto-increment/decrement addressing modes). Variables are syntactic abstractions of addresses in the store.

Languages support two ways of programming aggregates. The first, or *element-wise* method, relies on using *selection* and *update* operations to retrieve an element from or to replace an element in an aggregate. This method of programming with aggregates is characteristic of most imperative languages. These languages only define operations at the element-level so any operations on aggregates need to be specified in terms of individually retrieving, operating on, and updating the elements of an aggregate. For example, the following Fortran fragment increments the value of every element in an array `A` by `1`:

```
      DO 10 I = 1, N
10       A(I) = A(I)+1
```

The task of a parallelizing compiler is complicated by requiring the compiler to unravel all the dependencies on individual array elements to discover opportunities for parallelism. The success of the analysis performed by such compilers rests on being able to predict access and update patterns of the elements in an aggregate at compile-time. Such analysis is conventionally attempted in vectorizing compilers [Lampor74], [Polych86], [Burke86]. Automatic analysis is only partly successful in these cases and explicit programmer assistance is often required to achieve parallelism. The difficulty of extracting parallelism from languages which operate on aggregates at an element-level is a direct result of the word-at-a-time view of computation enforced by the effective width of the von-Neumann (processor-memory connection) bottleneck. For example, consider the following Fortran code:

```
      S = 0
      DO 10 I = 1, N
10       S = S + A(I)
```

Where S will yield the sum of all the elements in A. The presence of an anti-dependence on S in the recurrence equation means that no opportunity for automatic parallelism exists unless a compiler can re-write this inherently sequential code to yield a semantically-equivalent parallel rendition. This example is an instance of a *reduction* (where an array is reduced to a single element by repeated application of an associative operator to pairs of array elements). Reductions can be implemented in parallel with O(*log*(*n*)) complexity.

The alternative to the element-wise form of programming relies on providing *monolithic* operations which are defined on an entire aggregate (e.g. SUM, which produces the sum of all elements, or TRANSPOSE, which transposes elements in a matrix). The advantage of the monolithic approach is that the semantics of a monolithic operation are well understood and the access patterns of elements are *a priori* known which allows much more parallelism to be extracted. A strong argument in favor of the monolithic style of programming is made in [Backus78]. A disadvantage of the monolithic approach is that it requires that all monolithic operations be pre-defined, as part of the built-in set of operations provided by the language. This is the approach taken by latest Fortran standard, Fortran 90 [ISO89]. In first-order languages (such as Fortran) this leads to a proliferation of monolithic operations to take into account every eventuality. The introduction of higher-order monolithic operations which are parameterized by user-defined functions extends the scope of the monolithic approach and limits the proliferation of operations. The equivalence between some of the reduction operations provided in Fortran-90 and instances of the higher-order reduction operation `reduce` is given below:

Fortran-90	Higher-Order Reduction
ALL	reduce (∧)
ANY	reduce (∨)
PRODUCT	reduce (*)
SUM	reduce (+)

The previous code example which incremented every element of A by 1 can be performed by another higher-order monolithic operation, `map`:

```
map (λx. + x 1)
```

where the lambda-expression `(λx. + x 1)` is a function which takes a single argument (x) and adds 1 to it. The semantics of `map` specifies that the function parameter is applied to every element independently (and in parallel). Languages restricted to first-order monolithic operations are unnecessarily verbose and complex. The effectiveness of integrating monolithic operations in a language such as Fortran, which epitomizes the design of languages assuming a von-Neumann execution model, is limited (see [Bernec89]).

One of the greatest advantages of monolithic operations is that the complexity and run-time performance of programs using these primitive operations can readily be determined. Monolithic operations exhibit *transparency* (predictable performance by inspection) in the same way that the primitive operations in sequential languages have known and predictable run-time behavior.[1] Transparency ensures that inefficient aspects of programs can be identified syntactically and allows the programmer to develop intuitions for optimizing the performance of programs.

1.2. Rationale

This thesis aims to explore the development of a language for data-parallelism which takes into account the need for higher-order monolithic operations, enforces a strict notion of locality, provides constructs to perform parallel communication, and allows for the elegant and concise description of data-parallel algorithms. It is argued that these requirements are most adequately met by a functional language. In the remainder of this section the design criteria for a data-parallel language are explored in two ways: the first involves a "top-down," language-oriented, exposition of the data-parallel model of computation. This includes the issues of process- v.s. data-parallelism, algebraic data-types, monolithic operations, and arrays. The second way consists of a "bottom-up," machine-oriented description of parallel machine architectures and their capabilities. These two explorations serve to delimit the design space for a data-parallel language. This is because a language must present an adequate abstraction of target machine capabilities whilst exploiting performance potential to the full.

Exploiting parallelism in computation presents a significant problem in how work is to be divided up and distributed to independent processors for concurrent execution. There are two main approaches: (1) distribute the control-structure of the program, (2) distribute the data of the program. In the first approach, control constructs (procedures, functions, etc.) are allocated to processors and data is fetched when required by the particular task allocated to a processor: the operands come to the operators. In the second approach operators are sent to the operands.

1.2.1. The Case for Functional Languages

Our interest in functional languages is motivated by the following factors:

- The presence of referential transparency.

- The lack of a globally-accessible/modifiable state.

---

[1] Basic arithmetic/logical operations are usually assumed to be constant-time operations. These assumptions are of course affected by a variety of machine-specific aspects (e.g. presence of caches).

- The existence of opportunities for the automatic exploitation of implicit parallelism.

- The expressiveness provided by higher-order functions and polymorphism.

- The existence of well-developed transformation technologies which allow programs to be transformed via the use of meaning-preserving operations.

Issues such as side-effects, non-determinism, and synchronization are major impediments to the use of imperative languages for parallel programming. Functional languages are higher-order, do not rely on a global state, and do not allow destructive assignment. As a result, programs written in functional languages are expressive, side-effect free, and deterministic [Hudak89] [Meerte86]. All of these attributes are a considerable advantage to exploiting a data-parallel model of computation in a functional language.

1.2.2. Process-Parallelism

Process-parallelism consists of a number of independent threads of control engaging in independent, concurrent, computation. When data-structures are shared between processes, communication or explicit synchronization and exclusion mechanisms (via semaphores, monitors, or other schemes) may be necessary to ensure determinacy.

The parallelism inherent in functional languages is a result of the Church-Rosser theorem which states that expressions can be reduced in any order and still yield a unique result if the evaluation of the expressions terminates. The arguments to any operator or function can therefore be evaluated in parallel. A graph of dependent tasks can be maintained which makes all synchronization implicit: tasks execute when their results are demanded. This execution model is called *graph reduction*. Each individual expression can become a task to be spawned off and executed on a remote processor. Graph-reduction yields fine-grained processes. To date, all parallel implementations of graph-reduction have employed the process-parallel model of computation (e.g. Alice [Cripps87], GRIP [Peyton87a], <v, G> machine [August89], FLAGSHIP [Watson89]). The performance of these implementations has been unsatisfactory because of the excessively fine granularity of processes, high execution overheads, lack of control over locality, and the inefficient use of aggregate data-structures (arrays in particular).

To reduce the overheads involved with fine-grain processes, coarse-grain reduction models, such as Goldberg's Serial Combinators [Hudak85], and Gaudiot's

Macro-Actor [Gaudio85] have been developed. All data required by a spawned task needs to be fetched on-demand by the remote processor from other processors which hold the data. The process-parallel model of computation holds control to be more important than data. The grain-size of processes, not the availability of needed data, becomes paramount: granularity is given primacy over locality. This is unfortunate because it is the latency incurred by communication which often wipes out any benefit in distributing computation. Communication latencies in existing parallel machines are typically between one and two orders of magnitude higher than memory latencies.

The most serious indictment of a process-parallel model of computation is that it neglects the important issue of locality by making data subsidiary to control (i.e. operands are sent to the operators). The use of data-structures often results in the communication of large quantities of data.

1.2.3. Data-Parallelism & Monolithic Operations

Exclusive pursuit of the process-parallel model of computation for functional languages has meant that almost no research on the exploitation of a data-parallel model of computation for functional languages exists, despite the ready availability of high-performance and cost-effective data-parallel architectures (e.g. vector- and array-processing machines). Even Hudak's implementation of a functional language on an array-processor entailed simulating a process-parallel model of computation (with predictably poor results) [Hudak88]. The data-parallel model solves the problems of task granularity (because control follows data and distribution of control is faster than distribution of data) and provides machine support for efficient use of aggregate data-structures. Functional languages bring additional benefits to the data-parallel model through their inherent support of monolithic operations via higher-order functions, algebraic data-types, and referential transparency.

Data-parallel computation consists of applying an operation to a number of data elements concurrently. The *instantaneous degree of parallelism* is defined as the number of potentially parallel tasks that are created as a result of evaluating a single expression (e.g. the evaluation of $E\ (x, y) = f\ x + g\ y$ in a graph-reduction system creates two potentially parallel tasks, one to evaluate $f\ x$, the other $g\ y$. The instantaneous degree of parallelism is therefore *statically* determined by the program form. In contrast, a data-parallel operation such as `map f ar` which indicates that `f` can be applied ("mapped") to all elements of the aggregate `ar` simultaneously. Data-parallel operations therefore exhibit an instantaneous degree of parallelism only determined at *run-time* by the number of data elements in an aggregate or the number of available processing elements. Because the same operation can be broadcast to all elements, computation at all PEs will start and terminate at the same

time. The realization of data-parallelism is therefore inherently *synchronous* and no additional synchronization is necessary.

Data-parallel algorithms often consist of many lightweight *threads* (a small sequence of simple instructions). The ability to exploit fully the instantaneous degree of parallelism becomes important because of the short lifetime of these threads. If there is a non-trivial startup delay, or *latency*, between one independent thread and the next one, then the actual degree of parallelism realized by the execution of a program will never be optimal.

1.2.3.1. Algebraic Data-Types (ADTs)

Functional languages contain substantial support for the definition and manipulation of user-defined aggregate data-structures. User-defined data-types can be defined as algebraic data-types:

Definition 1.1 An *algebraic data-type* D, with type variables $\alpha_1, \ldots, \alpha_n$ is an expression:

$$D(\alpha_1, \ldots, \alpha_n) = k_1 \mid \ldots \mid k_l \mid c_1\, \tau_1 \mid \ldots \mid c_m\, \tau_m$$

where $k_1 \ldots k_l$ are constants, $c_1 \ldots c_m$ are constructors and $\tau_1 \ldots \tau_m$ are of the form:

$$\tau_i \in \{\alpha_1, \ldots, \alpha_n, D'(\tau_p, \ldots, \tau_q)\}, \qquad 1 \le i \le m$$

where $D'(\tau_p, \ldots, \tau_q)$ is a type.
□

Constructors are labels to distinguish different components of the data type. For example lists and trees are data-types which can be defined as ADTs:

```
list α = nil | cons(α × list a)

tree α = empty | leaf α | node(tree α × tree α)
```

Where × is an infix type-constructor used to denote the cartesian product. In this example, `nil`, `cons`, `empty`, `leaf`, and `node` are constructors. These definitions are *polymorphic* in that they permit lists and trees of any type (α) to be defined. A list of integers [1, 2, 3] can be constructed using the expression:

```
cons(1, cons(2, cons(3, nil)))
```

Algebraic data-types permit monolithic operations to be deduced *automatically* from the definition of the data-type. It is the availability of ADTs and their higher-order capability that gives functional languages inherent support for monolithic operations. This duality between algebraic data-types and monolithic operations relies on a semantics based on the notions of categories and functors.

A *category* consists of objects and functions on those objects. A *functor* is a function from one category to another. A functor therefore has two aspects: (1) a mapping from objects to objects, and (2) a mapping from functions to functions. For example, the type-constructor '×' is a component of a functor which takes objects of one category to another. The other component of this functor takes functions and is defined as:

```
f × g = λ(x, y). (f x, g y)
```

which takes two functions, `f` and `g`, and a pair of values `(x, y)` as arguments and returns a pair consisting of `f` applied to `x` and `g` applied to `y`. Similarly, for a specific algebraic data-type, the data-type definition is the object part of a functor. The remaining part of the functor is automatically produced from the data-type as follows:

Definition 1.2 The monolithic `map` function for an algebraic data-type D (with type-variables $\alpha_1 \ldots \alpha_n$) is defined as:

$$
\begin{aligned}
D(f_1, \ldots, f_n) = {} & \lambda k_1.\ k_1 \\
& \mid \ldots \\
& \mid \lambda k_l.\ k_l \\
& \mid \lambda c_1\, x.\ c_1((\tau[\![\tau_1]\!])\ x) \\
& \mid \ldots \\
& \mid \lambda c_m\, x.\ c_m((\tau[\![\tau_m]\!])\ x)
\end{aligned}
$$

$$
\begin{aligned}
\tau[\![D'(\tau_p, \ldots, \tau_q)]\!] &= D'(\tau[\![\tau_p]\!], \ldots, \tau[\![\tau_q]\!]) \\
\tau[\![\alpha_i]\!] &= f_i
\end{aligned}
$$

This function is a functor because:

$$
\begin{aligned}
D\ id &\equiv id \\
D\ f{\circ}g &\equiv (D\ f){\circ}(D\ g)
\end{aligned}
$$

Where *id* is the identity function and ($\circ$) denotes function composition.

□

For every element of type-variable α_i, this function applies an element f_i. It leaves the underlying structure of the data object unchanged. The f_i functions could be applied to all elements of the aggregate concurrently and independently of any other elements in the aggregate.

The `map` function on lists is therefore:[2]

```
list f = λnil. nil
       | λcons p. cons((f × list f) p)
```

[2]In category theory it is conventional to give the function-component of a functor (`map`) the same name as the data-component (i.e. the algebraic data-type).

Evaluating the × function yields the more conventional definition:

```
list f = λnil. nil
       | λcons (h, t). cons(f h, list f t)
```

Similarly for trees:

```
tree f = λempty. empty
       | λleaf x. leaf (f x)
       | λnode p. node (((tree f) × (tree f)) p)
```

The function `f` can be applied to all elements of the list and all leaves of a tree concurrently. The semantics of the `map` functor permits an implementation whereby the function argument is applied to all elements of a data-structure in parallel. This provides a semantic framework for the formal development of data-parallel operations.

All implementations of algebraic data-types use pointer-based structures whose components are allocated dynamically, as required. This precludes instantaneous, parallel, and distributed access to data-elements, as required for the exploitation of data-parallelism.

The categorical model serves to provide a sound semantic foundation for monolithic programming, but the realization of data-parallelism requires an implementation which uses aggregate data-structures which permit simultaneous access to all data-elements. All of the processors in a data-parallel machine need to be able to access their allotment of data-elements instantaneously, without waiting for other processors.

Arrays are data-structures which are allocated in their entirety upon declaration. This permits the elements of an array to be evenly distributed across all available processors, which in turn permits instantaneous access to all elements in parallel. Arrays are isomorphic to lists so the categorical foundation for monolithic operations can be used to provide a semantics for monolithic operations on arrays.

1.2.3.2. Arrays

Arrays have traditionally been a weak point in sequential implementations of functional languages. To enable efficient access, they are allocated as contiguous segments of memory. To retain referential transparency, updating a single element of an array requires a copy of the entire array to be made. Numerous solutions to circumvent this inefficiency have been proposed: non-referentially transparent arrays (i.e. as in ML), reference-counts (either static, at compile-time [Bloss89], or at run-time [Hughes87] [Glaser88]), psuedo-array implementations (via trailer-lists [Aasa88], as trees), special types (linear type theory) [Wadler89]. Haskell [Hudak90], a new functional language, includes *incremental arrays* where an array is declared by a list of pairs: (*index*, *value*) with the understanding that each index only occurs once. Incremental arrays are designed to combat the problem of space inefficiency but sacrifice parallelism (since access to elements of the update list is

constrained by sequential access to list elements). Each of these solutions has been less than satisfactory and the "array-update" problem remains an open research area.

The array is the most suitable aggregate for the realization of data-parallelism and the use of monolithic operations on data-parallel architectures solves the cost problem of array updates. Updating an array on a data-parallel machine takes constant time without loss of referential transparency (since the distribution of array elements ensures that all processors can copy all of the elements in an array in parallel).

1.2.4. Parallel Architectures

Parallelism can only be exploited by a suitable combination of software and hardware. The process- and data-parallel models of computation are supported by corresponding machine architectures. Flynn's [Flynn72] taxonomy of parallel machines distinguishes between two main classes of parallel machine architectures: MIMD & SIMD.

Multiple-Instruction-Multiple-Datastream (MIMD) systems correspond to platforms designed for the exploitation of process-parallelism. MIMD systems consist of powerful interconnected processors, each capable of independently executing a task. Such systems are further sub-divided into shared-memory, distributed-memory, and fixed-interconnectivity, or general-interconnectivity systems. The flexibility provided by having each processor execute its own (possibly distinct) program is offset by the potentially high costs of message-passing, synchronization, and task-allocation.

Single-Instruction-Multiple-Datastream (SIMD) systems are expressly designed to exploit data-parallelism. Typically, these systems consist of a *central processing unit* (CPU) and thousands of simple interconnected *processing elements* (PEs). The CPU is equivalent to a conventional sequential processor but can also broadcast instructions to the PEs for concurrent execution. Every PE executes the same, globally-broadcast, instruction stream. Two variants of the basic arithmetic/logical instructions exist: those that operate on ordinary scalar values (in central, scalar memory accessible to the CPU) and those that operate on locations in the memory of individual PEs. Each PE accesses the local contents of the same, globally-broadcast, operand address. These instructions can be viewed as executing on *planes* of memory consisting of the same addresses in distributed PE stores.

Vector-processing systems may be considered a sub-class of SIMD architectures.[3] Machines such as the Cray Corp. Cray I and the Control-Data Corp. Cyber are good examples. Two-dimensional vector-processing systems are called *array-processors*. Early examples of this type of architecture are the Goodyear MPP [Potter85], and AMT DAP [Active88]. More recent systems such as the Thinking Machine Corp. Connection-

[3]Vector-processors lack distributed memories and therefore suffer from the restrictions of the von-Neumann bottleneck in transferring vectors in and out of a central global store.

Machine [Hillis87], and MasPar MP-1 [Blank90] include more powerful processing elements and general interconnection networks to implement non-local message-routing.

The main attractions of SIMD architectures are:

- Synchronous operation.
- Efficient communication facilities.
- Scalable performance.

The broadcast capability means that instantaneous parallelism in the order of the number of available processors is available at the cost of decoding a single instruction. The synchronous nature of SIMD machines means that there is no start-up delay involved in data-parallel computation and no synchronization costs are incurred in waiting for processors to finish executing their alloted instructions. The instantaneous degree of parallelism exploited by SIMD machines is constrained only by the number of processing elements or the number of data-elements in an aggregate. There is no need to specify process granularity or the distribution of data. The fact that all the PEs execute their instructions synchronously means that no explicit synchronization to guarantee determinacy is necessary.

The PEs in conventional SIMD machines are locally interconnected through high-speed fixed links. Arrays provide a suitable abstraction of locally-interconnected distributed PE stores. Some newer architectures also provide general, non-local, communication capabilities in the form of routing networks (e.g. MasPar MP-1, Thinking Machines CM-2). The distinction between these two forms of interconnection lies in that local interconnections use dedicated links whereas global interconnections require links to be shared with adverse effects with regard to link contention. The performance of non-local communication over shared links is unpredictable. Local communication facilities are contention-free and therefore have predictable performance.[4]

These rich interconnection facilities allow parallelism to be exploited in the concurrent movement of data elements. The inability of SIMD systems to support parallel independent threads of computation on PEs dictates that exploitation of parallelism inherent in data-movement is important, which is why most existing SIMD architectures invest heavily in inter-PE communication facilities (e.g. the fully-configured Connection Machine features 16 communication links per PE [Thinki87]).

The fact that computation occurs across distributed data elements (and the inherent support for broadcast and synchronization) means that computation on SIMD architectures

[4]On the Connection Machine and MP-1 architectures local communication facilities are roughly two-orders of magnitude faster than global communication.

is inherently scalable as the only restriction on parallelism is the number of data elements or PEs.

1.3. Difficulties with the Data-Parallel Model

Despite the numerous advantages of the data-parallel model of computation, a number of impediments restrict its wider use. These impediments fall into three categories:

- Limitations of existing hardware platforms.
- Conceptual problems which arise from a predominantly sequential mode of decomposing computation.
- Lack of expressiveness in existing language which make exploitation of data-parallelism unnecessarily difficult.

SIMD machines are the predominant class of architectures supporting the data-parallel model of computation. The restriction of a single flow of control in these machines means that languages for SIMD machines are constrained by this restriction. Improvements in technology and greater experience with data-parallelism will lead to better architectures which impose fewer restrictions on languages.

The conceptual difficulty which arises from data-parallel programming is because programmers are most familiar with sequential architectures based on the von-Neumann model. The von-Neumann model encourages a word-at-a-time view of computation. This results in a sequential and scalar approach to the decomposition of computation. Data-parallel operations are naturally parallel and monolithic. This requires a "global" instead of a "local" view of computation and control. The conceptual difficulty can be eased by a language with sufficient expressive power and a natural set of data-parallel opeations which complement the underlying language paradigm.

Acceptance of the data-parallel model has been severely hampered by the lack of effective languages for data-parallel programming. Adaptations of sequential imperative languages have serious shortcomings which have helped to convey an overly pessimistic view of the difficulty associated with exploiting data-parallelism. The work presented in this thesis is an attempt to address some of the limitations of these adapted sequential languages.

1.3.1 Limitations of SIMD

SIMD realizations of the data-parallel model of computation suffer from the following restrictions:

- The parallelism exploited is "horizontal" (i.e. concurrent application of the same operation) instead of "vertical" (i.e. composition of different computational stages). Vertical parallelism is also known as *pipeline parallelism* and results from a natural decomposition of a problem into a sequence of dependent tasks.

- It is difficult to cope with heterogeneous data (i.e. data-elements of different types). Heterogeneity introduces conditional statements (used to test data-elements, so that type-dependent computation can be carried out) which in a SIMD machine requires each branch of a conditional statement to be executed in turn, sequentially. Parallelism is lost as a result.

- SIMD computation on hierarchical data is inefficient (e.g. graphs). Hierarchicies (i.e. dependencies) in data-elements require that the different levels of the hierarchy be traversed in turn, sacrificing parallelism.

More powerful and general forms of data-parallel architectures where the restriction on a single flow of control is relaxed can help to circumvent these limitations. These difficulties are symptomatic of the current state of technology and are not indictments of the data-parallel model of computation.

1.3.2. Conceptual Difficulties with Data-Parallelism

Conceptual difficulties encountered by programmers in decomposing algorithms for data-parallelism include:

- It is difficult to specify naturally asychronous algorithms. The data-parallel model requires a global viewpoint which is often unsuited for algorithms which solve a problem by decomposing it into smaller sub-problems, each of which can be solved independently (e.g. a divide-and-conquer approach). It is difficult to reconcile monolithic operations with the notion of decomposition of data and control.

- The strict enforcement of locality makes it necessary to specify communication in addition to computation. In sequential languages communication is implicit.

Programming with monolithic operations requires a change in viewpoint on behalf of the programmer. The fact that the monolithic approach is unfamiliar to most programmers means that a new approach to solving problems needs to be learned. This

learning process is facilitated by powerful and natural languages. Naturally asynchronous algorithms which rely on dividing the set of data-elements into subsets are difficult to specify in the monolithic style. It may be possible to present common asynchronous algorithms as higher-order functions which facilitates a monolithic implementation. This remains an open area of research. The strict enforcement of locality requires the programmer to specify communication as an extra component of an algorithm. In imperative languages this is unnecessary so the specification of communication is an extra burden to the imperative programmer. The availability of pre-defined communication operations eases the burden and aids in showing the way to a parallel solution.

1.3.3. Shortcomings of Adapted von-Neumann Languages

Some of the difficulties encountered in programming data-parallel machines can be ameliorated by providing sufficiently powerful languages. Adaptations of sequential languages have the following shortcomings:

- Too many monolithic operations due to first-order restrictions of the language.

- Confusing semantics because sequential and parallel parts of the program are not clearly separate. This is particularly true where monolithic operations consist of overloaded versions of the basic arithmetic/logical operations.

- Lack of parallel conditional statements which makes it difficult to deal with heterogeneous data-elements or hierarchies in data.

- Lack of general operations to perform communication. This makes it difficult to comply with the strict enforcement of locality.

Addressing these shortcomings requires a language which is considerably more high-level and expressive than existing data-parallel languages. The high level of abstraction provided by a higher-order data-parallel language based on the functional style requires considerably more investment on the part of the compilation scheme and run-time system. Support for higher-order functions, algebraic data-types, and recursive functions in a data-parallel model of computation requires new and sophisticated compilation techniques.

1.4. Philosophy of Approach

The list of requirements for a language for data-parallel programming have been shown to include:

- Lack of global state.
- Monolithic operations on an aggregate data-type.
- Higher-order capability.
- Language constructs to express communication & enforce locality constraints.

Functional languages fit all of the above criteria and furthermore possess a rich foundation of aggregate data-types based on the notion of algebraic data-types. Monolithic operations are the natural complement of algebraic data-types. A data-parallel model of computation makes monolithic operations on arrays efficient without loss of referential transparency. This suggests that a language consisting of a functional language augmented by a set of primitive monolithic operations defined on arrays forms a natural vehicle for the expression of data-parallelism.

1.4.1. Design Principles

The choice of primitives and their implementations are guided by the principles of *transparency*, *locality*, and *uniformity*. Transparency is inherent in monolithic operations in a data-parallel model of computation.

Transparency yields predictable performance and aids the development of intuitions about program efficiency. Transparency is of paramount importance in the development of optimizations and program transformation methods.

Locality is related to the speed of access to data-elements. When memory is distributed, the availability of operands in local store is crucial to performance. Throughout this thesis, a canonical allocation of array elements to processing elements is assumed, whereby each PE is allocated a single element of an array. The term locality is used to denote two related ideas: (1) *intra-element locality*: elements at the same index in conformant arrays are assumed to be mapped to the same PE memory and therefore exploit locality in that an operation can be applied to both elements without communication. (2) *inter-element locality*: elements whose indices are logically-adjacent are physically-adjacent. Physical adjacency in this case relates to the availability of fast, contention-free access rather than the spatial distribution of neighboring elements. Access to adjacent

elements is significantly faster than access to other elements. Computation which requires access to logically-adjacent data-elements exploits inter-element locality. A language for data-parallelism which includes communication constructs should respect the notion of inter-element locality and provide means of distinguishing between local and non-local communication. Computation without communication is the ideal. If communication is necessary, it must be performed as efficiently as possible.

The principle of uniformity dictates that a given primitive should have only one operational semantics. This principle leads to a number of restrictions being imposed on arrays, as will be described in Chapter 2.

1.4.2. Bias Towards SIMD

SIMD machines are the most common form of data-parallel machines. The development of the abstract data-parallel machine (PAM) in Chapter 4 as a target for compilation is therefore biased towards an abstract form of commercially-available SIMD architectures. This close correspondence has been chosen explicitly to facilitate the mapping of the abstract machine onto the capabilities of real data-parallel machines.

SIMD machines have more in common with ordinary sequential machines than MIMD machines hence it is entirely appropriate that the compilation scheme is developed by adopting an existing, sequential, abstract machine compilation method as a starting point. The reader can gain familiarity with techniques for compiling functional languages for conventional sequential architectures before the compilation scheme for PAM is developed. The compilation strategy for the abstract data-parallel machine can then be compared and contrasted with that of the conventional sequential machine.

1.4.3. A Hybrid Approach to Transformation

The simple underlying machine model (PAM) makes it easy to identify potential sources of inefficiency. The inefficiencies in generated code can be removed by compiler optimizations or by program transformation methods. The lack of an inductive definition for the array data-type or the data-parallel primitives means that transformation of data-parallel functional programs requires an algebraic style of transformations based on axioms. The unfold/fold transformation methodology which is more commonly used in functional programming can still be applied to the transformation of conventional functions used as arguments to the higher-order monolithic operations. These two transformation methods therefore complement each other: the algebraic style can be applied to compositions of monolithic primitives, and the unfold/fold methodology to the transformation of individual function arguments (of monolithic operations).

1.4.4. Objective

The goal of this thesis is to demonstrate the elegance of the functional style of data-parallel programming and to illustrate how programs written using data-parallel primitives can be transformed, optimized, and compiled for an abstract data-parallel architecture.

1.5. Statement of Originality

The original contributions of this thesis are as follows:

- Identification of a core set of data-parallel primitives to express computation and communication and the demonstration of how other common parallel monolithic operations can be defined in terms of the initial primitives.

- Specification of an abstract data-parallel machine architecture.

- Development of a compilation scheme from the extended functional language to abstract data-parallel machine code which preserves the higher-order capability, recursion, arbitrary control, and algebraic data-types available in the underlying language.

- Development of support for a normal-order reduction strategy employed in PAM and analysis of the advantage of this reduction strategy in the context of existing data-parallel machine architectures.

- Identification of the application of existing compiler optimization techniques to data-parallel implementations of functional languages (strictness analysis, sharing analysis).

- Development of new compiler optimizations (elimination of bifurcating control-flow, termination-testing, iteration-lifting) applicable to data-parallel implementations of functional languages.

- Extension of existing transformational approaches to data-parallel program development.

- Adaptation of existing work for the derivation of efficient forms of communication via transformation and the development of extensions to

include the promotion of computation into communication to aid the derivation of efficient data-parallel program forms.

1.6. Thesis Overview

In Chapter 2 a simple functional language augmented with data-parallel primitives is defined which will be used for all example programs throughout the thesis. The core set of primitives allows for the expression of a rich set of derived operations consisting of simple functions defined in terms of the original primitives. Possible alternative aggregates for data-parallel programming are outlined and realization of these alternatives in terms of an underlying array representation is demonstrated.

In Chapter 3 this extended functional language is used to program a variety of small applications. These examples illustrate how the language can be used to solve "real" problems. The resulting programs are clear, concise, and provide ample opportunity for the exploitation of data-parallelism in both communication and computation. All of the sample programs have been run using sequential implementations of the primitive and derived operations.

The remainder of the thesis is concerned with the issue of compilation; in particular, how our language can be compiled to a data-parallel architecture. Rather than develop a compilation strategy for a particular machine, an abstract machine architecture is developed by extending a common abstract machine model used in the compilation of functional languages for sequential machines. The original sequential model is explained in Chapter 4 followed by a discussion of the modifications necessary to abstract the capabilities of data-parallel machines. In Chapter 5 the full abstract data-parallel machine (PAM) is introduced, characterized by its abstract instruction set defined by transitions on the machine state. The compilation scheme translates the programs written in the extended functional language into sequences of PAM-code. The PAM code for each of the compilation examples in the thesis has been produced automatically by a code generator which implements the compilation rules described in this chapter.

The compilation scheme is explained by concentrating on the aspects which are unique to PAM. In particular these involve the translation of control-structures (conditional-statements, recursion), algebraic data-types, and suspended evaluations. In Chapter 6 the compilation scheme is illustrated by compiling some simple code fragments and explaining the run-time behavior of particular aspects of the abstract machine instructions. This discussion leads to the exploration of efficiency issues and an attempt to qualify the notion of efficiency in the context of a data-parallel machine such as PAM. Potential inefficient aspects of PAM are explained.

The issue of optimization is addressed in Chapter 7. Optimizations fall into two classes: automatic and user-assisted. The automatic optimizations can be performed by the compiler and rely on being able to identify inefficient code sequences. A number of optimizations used for sequential implementations of functional languages can be used but the particular execution model used by PAM introduces new sources of inefficiency. New optimization techniques are developed to deal with some of these. The user-assisted optimizations are more general and involve the use of source-to-source transformations which exploits the fact that our data-parallel language is referentially-transparent (due to the freedom from side-effects). Existing transformational methods are also applicable. A number of axioms and lemmas for the operations defined in Chapter 2 are presented. A novel use of transformation for optimizing communication is then introduced which is derived from the work in Parallel Data Transforms (PDT) [Flande87]. The PDT system is applied in the context of our functional data-parallel language and is extended to include the promotion of computation into communication which allows additional parallelism to be introduced via transformation. The transformations introduced in this chapter give rise to the possibility of deriving a particular class of parallel communication-efficient algorithms (systolic algorithms) from initial, non-systolic formulations through step-wise refinements.

In Chapter 8 the omitted issues and further work are discussed. The work in this thesis is compared and contrasted with the work in the literature dealing with the exploitation of data-parallelism. Conclusions on the suitability of a functional language for programming data-parallel machines are presented.

Chapter 2

2. A Functional Language with Data-Parallel Operations

This chapter specifies a set of data-parallel operations as an extension to a simple functional language. This is an extension in the sense that although the operations could be specified in the lambda-calculus which is the underlying operational model of all functional languages, their parallel semantics is not adequately captured by it.

Two constraints must be satisfied by any language extensions to exploit architectural features: the extensions must be consistent with the existing language paradigm and fully utilize the abilities of the target machine. Additionally, it is desirable that the extensions be few and general.

The proposed extensions consist of a suite of built-in data-parallel operators defined on arrays. The array data-type is presented, followed by the definitions of the primitive operators. A set of derived operations is defined in terms of the built-in operators. Possible implementations of other aggregate data-types (bags and general algebraic data-types) is discussed and an underlying array representation for these data-types is suggested.

2.1. Introduction

Data-parallelism is inherently tied to the manipulation of aggregate data structures via a set of operations. These operations are monolithic in the sense that they apply to all of the elements of an aggregate. The aggregate data-structure forms a language-level abstraction of a mesh of locally-interconnected distributed memories. The most natural aggregate to represent such an abstraction is the array because there is a logical adjacency relationship between elements in an array based on their associated index values. Other aggregates can be realized in terms of an underlying array representation. Functional languages permit the creation of general-purpose monolithic operations on aggregate data structures.

2.2. A Lazy, Higher-Order, Functional Language

Our starting language is an extended version of the lambda-calculus in which programs are finite terms of the grammar given in fig. 2.1. This language is a syntactically-sugared form of the lambda-calculus in which all patterns on the left-hand side of function definitions have been lifted-out into explicit tests using conditional statements. This language is a common intermediate form used by functional language compilers. By adopting this intermediate form language-specific syntactic features are absent and the compilation scheme is simplified.

The Named Lambda-Calculus

	x	Variable or constructor
	c	Constructor
	k	Constant
	p	Primitive operation

$$
\begin{array}{lll}
e & ::= & k \\
 & | & c \\
 & | & p \\
 & | & x \\
 & | & e_1\ e_2 \\
 & | & \lambda x.e \\
 & | & \texttt{let}\ x_1 = e_1\ \texttt{in}\ e_2 \\
 & | & \texttt{case}\ e\ \texttt{in}\ a \\
a & & [\texttt{when}\ c\ x_1 \ldots x_m : e_1;]^+\ [\texttt{otherwise} : e_1] \\
p & ::= & \ | - | * | / | = | \neq | \wedge | \vee
\end{array}
$$

fig. 2.1

For conciseness, programs written using this simple language may use the following notation:

$$
\begin{aligned}
f \circ g\ x &\equiv f\ (g\ x) && \textit{(function composition)}\\
f\ \hat{}\ n &\equiv f \circ f \circ \ldots f && \textit{(number of } f = n\textit{)}\\
\texttt{hd}\ [x_0, \ldots, x_n] &\equiv x_0 && \textit{(returns the head of a list)}\\
\texttt{tl}\ [x_0, \ldots, x_n] &\equiv [x_1, \ldots, x_n] && \textit{(returns the tail of a list)}\\
p \rightarrow q; r &\equiv \texttt{case}\ p\ \texttt{in true : } q\texttt{; false : } r && \textit{(shorthand conditional)}\\
\lambda(x_0, \ldots, x_n).\ e &\equiv \lambda p.\ \texttt{case}\ p\ \texttt{in when}\ \textit{tuple}_n\ x_0 \ldots x_n\ \texttt{:}\ e && \textit{(tuple-input)}\\
\lambda x.\ (e_0, \ldots, e_n) &\equiv \lambda p.\ \textit{tuple}_n\ e_0 \ldots e_n && \textit{(tuple-output)}
\end{aligned}
$$

The ($\hat{}$) notation is used to denote iteration. Similarly, $p \rightarrow q; r$ is merely a shorthand to denote the common boolean conditional-statement. Tuples are an instance of algebraic data-types with built-in constructors, $\textit{tuple}_i \ldots \textit{tuple}_j$ corresponding to the arity of the tuple.[1] According to whether tuples are used as inputs or returned as outputs, there is an equivalent form in the syntax given by fig. 2.1.

Functions are first-class citizens and lazy evaluation (normal-order reduction to weak-head normal-form (WHNF) with sharing) is used for the reduction of expressions. Performing normal-order reduction is more difficult than applicative-order reduction as the compilation is complicated by the need to build delayed evaluations (closures). A normal-order reduction scheme seems an incongruous choice for a language to explore data-parallelism. There are several reasons why it is attractive:

- To explore the restrictions imposed by a data-parallel architecture on a normal-order reduction strategy.

- To pursue the development of a lazy, data-parallel, programming style.

- Lazy languages allow a more expressive style of programming, are essential for declarative I/O, and are the growing consensus in the functional programming area.

2.3. The Array Data-Type

In this thesis arrays are strict, homogeneous, aggregate data structures. All elements of an array must therefore be evaluated at the same time and must all be of the same type. Another way of specifying that arrays are strict is to say for an array B: $B = \bot$ iff $(\exists\ b_i \in B, b_i = \bot)$. Array elements belong to the domain D, specified below:

g	Ground type: *Bool, Int, Char, Real*
c	Constructor

$$D ::= g \mid D \rightarrow D \mid D \times D \mid c \mid c(D) \mid (D \mid D)$$

[1] Constructors cannot be partially-applied. All arguments must be present.

Note that functions are valid members of D. Array elements can be accessed by their location or *index*. An index i is either an integer or a tuple of integers. Two problems arise in the abstraction of local memories of processing elements as array data-structures: (1) how are multi-dimensional arrays allocated to processing elements? (2) How are *nested* arrays (arrays whose elements are arrays) supported? The answers to these questions are determined by the constraints of the target architecture and the desire to uphold the design principles of transparency, locality, and uniformity.

2.3.1. Multi-Dimensional Arrays

Allocating a multi-dimensional array to a machine raises the question of which dimensions are mapped *across* the PEs and which dimensions are mapped *into* the PE memories. This can lead to the loss of both transparency and the principle of uniformity (i.e. not all dimensions are equivalent). A data-parallel operation applied to such an array will be parallel in only some of its dimensions. Copying the elements of such an array no longer takes constant time. The time required is proportional to the number of elements in the sequential dimensions.

Alternatively, the array can be "flattened" so that elements in the extra dimensions are also distributed across the PEs. In this case uniformity is preserved at the expense of inter-element locality: elements with adjacent indices are no longer all physically adjacent. Some elements will still be physically adjacent while others will require non-local communication. Computation on elements with adjacent indices will have unpredictable performance.

Neither uniformity nor locality needs to be sacrificed if the dimensionality of arrays is restricted to the *machine dimension*, where the machine dimension is defined as the number of dedicated interconnections between processing elements. The complexity of communication over links which are shared is unpredictable due to contention for links and hot-spots within the interconnection topology. Local (fixed-link) communication facilities are contention-free and therefore have predictable performance. It is therefore worthwhile to distinguish between these two forms of communication in a data-parallel language. For example, the CM-2 is a 16-dimensional machine in that each PE is locally connected to 16 distinct neighbours. Therefore, arrays of up to 16 dimensions can be mapped on to the CM-2 without sacrificing uniformity or locality.

2.3.2. Nested Arrays

The limitation on the array element data-type D is that arrays may not be nested. Allowing nested arrays allows for the expression of what Blelloch calls *nested parallelism* [Blello90], or data-parallel operations which apply other data-parallel operations to their elements. Many parallel machines can only exploit one level of parallelism. Allowing nested levels

of parallelism requires choosing between outer- and inner- levels of parallelism with often greatly varying performance. It also contravenes the dictum of uniformity in that all data-parallel operations in a program should exhibit parallel behaviour. Blelloch shows how nested parallelism can be compiled into flat, single-level parallelism but at the expense of inter-element locality as above. Nested parallelism can be useful as a means of *expressing* hierarchically parallel computation. Supporting nested parallelism by allowing nested arrays would require either uniformity or locality to be sacrificed. A suitable compromise may be to allow nested parallelism in an initial problem specification but to remove any nesting by the application of program transformation techniques to yield a "flat" final program form.

2.4. Primitive Data-Parallel Operators

In proposing a set of built-in operations it is important that the operations be fully general. The idea of generality in a set of data-parallel operations is a difficult notion to define formally. Instead, it is possible to consider typical data-parallel machine architectures and provide abstractions that adequately exploit their capabilities. These become the built-in operations. Subsequent operations can then be defined in terms of this initial set with the assurance that they will adequately exploit available machine characteristics.

Data-parallelism can be exploited by any parallel architecture but the massive number of processing elements, broadcast capability, and high-bandwidth interprocessor communications offered by single-instruction, multiple-datastream (SIMD) machines make them the most natural platforms for exploiting data-parallelism. SIMD machines are purpose-built for the exploitation of data-parallelism and have become synonymous with a hardware realization of data-parallelism.

SIMD machines contain hardware to perform concurrent data-movement and global instruction-broadcasting. The design of the basic operators is therefore motivated by the desire to capture and exploit these two basic forms of data-parallelism.

Only one primitive operation to perform data-parallel computation is provided: `imap`. The type signature of `imap` is followed by its definition:

`imap : (Int` $\rightarrow \alpha \rightarrow \beta$`)` $\rightarrow$ `array(Int,` α`)` $\rightarrow$ `array(Int,` β`)`

`imap` $f \ll x_0, \ldots, x_n \gg \Rightarrow \ll f\ 0\ x_0, \ldots, f\ n\ x_n \gg$

The primitive takes a function and an array as arguments. The function parameter takes the index (`Int`) and an array element (α) and returns a new array element (β). The array data-type `array(Int,` α`)` denotes a one-dimensional array with elements of type α and whose indices are integers. Similarly, the type of two-dimensional arrays is denoted by `array(Int` $\times$ `Int,` α`)` where the indices are pairs of integers. The definitions of the

primitive operations are given here for one-dimensional arrays only although the definitions for the higher-dimensional array variants can readily be deduced from the definitions given here. All of the operations "scale" to arrays of any dimensionality. Arrays are represented using the notation $\ll x_0, \ldots, x_n \gg$ where elements are subscripted with their index.

The `imap` operation is the only built-in higher-order function and is an abstraction of a global instruction-broadcast mechanism. All subsequent data-parallel operations are constructed from compositions of `imap` and the data-movement operations. This approach is in contrast with a language like APL, which presents the programmer with a number of second-order monolithic array operations [Iverso62] which *combine* both communication and computation in a single operation (e.g. inner and outer-product operations) and provide only elementary facilities to express communication (e.g. transpose) separately. The monolithic style of programming introduced by APL was adopted by the functional community by the introduction of FP, a second-order language with primitive map (α) and reduction (/) operations defined on sequences (heterogeneous lists) [Backus78]. The Paralation model [Sabot88] provides a higher-order operation, `elwise`, which is equivalent to `imap`.

Data-movement can be divided into two classes: *local* and *global*. Local communication involves sending a value to some fixed destination relative to the current location. This offset is the same for all locations. This form of communication can therefore exploit inter-element locality by using high-speed fixed communication links. On the other hand, the destination of a data element may correspond to its value or some other aspect of the local state. This requires a more general interconnection facility as the pattern of communication is potentially irregular and elements may be routed anywhere. This form of communication may be subject to contention due to sharing of communication links. The `rotate` operation implements the first form of data-movement whereas the `send` and `fetch` operations correspond to the second form:

```
rotate : Int → array(Int, α) → array(Int, α)
```

$$\texttt{rotate}\ i \ll x_0, \ldots, x_n \gg \Rightarrow \begin{cases} \ll x_{n-i+1}, \ldots, x_n, x_0, \ldots, x_{n-i} \gg & i > 0 \\ \ll x_{|i|}, \ldots, x_n, x_0, \ldots, x_{|i+1|} \gg & i < 0 \\ \ll x_0, \ldots, x_n \gg & i = 0 \end{cases}$$

```
send : array(Int, Int) → array(Int, α) → array(Int, list α)
```

$$\texttt{send} \ll i_0, \ldots, i_n \gg \ll x_0, \ldots, x_n \gg \Rightarrow \ll [x_r \mid i_r{=}0], \ldots, [x_s \mid i_s{=}n] \gg$$

```
fetch : array(Int, Int) → array(Int, α) → array(Int, list α)
```

$$\texttt{fetch} \ll i_0, \ldots, i_n \gg \ll x_0, \ldots, x_n \gg \Rightarrow \ll [x_{i_0}], \ldots, [x_{i_n}] \gg$$

`rotate` displaces all elements in an array by the same distance, specified as an offset from the index. Elements on the edge of the array "rotate" round in toroidal fashion. Fortran-Plus [Jenkin90] (used to program the AMT DAP) has a different `rotate` operation depending on the direction in which array elements are being displaced and the dimensionality of the array. The Fortran-Plus operations and the equivalent form expressed in terms of `rotate` are given below:

Fortran-Plus	Equivalent `rotate`
SHLC	`rotate -1`
SHRC	`rotate 1`
SHNC	`rotate (-1, 0)`
SHEC	`rotate (0, -1)`
SHWC	`rotate (0, 1)`
SHSC	`rotate (1, 0)`

The Fortran-Plus operations only permit the expression of a displacement of unit distance and along a single dimension at a time. Fortran 90 provides a single operation to express rotation of data-elements: `CSHIFT`. This operation applies to any array regardless of dimensionality but only permits rotation in a single dimension. The `rotate` operation applies to all arrays, regardless of dimensionality, and allows displacements of any distance along any number of dimensions to be expressed. For example compositions of several data-movement operations in Fortran-Plus can be expressed using a single `rotate`:

$$\texttt{SHNC}\circ\texttt{SHEC}\circ\texttt{SHEC} \equiv \texttt{rotate (-1, -2)}$$

The `rotate` operation is an abstraction of the capabilities provided by the fixed, nearest-neighbour interconnections available on all SIMD machines. Alternative abstractions are provided by two recent languages designed expressly for SIMD computation, DAPL [Rice89] and Parallaxis [Br91]. These languages permit the definition of topologies in a manner akin to the definition of user-defined data-types so that forms of communication can be named, providing an additional level of abstraction. DAPL permits the definition of topologies in terms of pre-defined *geometric types* which include grids, boxes, hypercubes, and binary trees. The Parallaxis system provides a more general system where the interconnections between elements are specified by the use of a `CONNECTION` directive which names the links between elements and thereby defines the topology.

The `send` and `fetch` operations are the most general communication operations. In `send`, each element in the second array is sent to a destination index contained in the corresponding element in the first array. Multiple elements arriving at the same destination are accumulated in a list. In `fetch`, each element of the first array contains the index of the element in the second array that is to be brought to the current location.

The `send` and `fetch` operations rely on the global routing interconnections available on more recent SIMD machines (e.g. MasPar MP-1 [Blank90] and the Connection Machine CM-2 [Thinki87]). These routing mechanisms may be non-deterministic. The operational behaviour of the general router imposes constraints on the use of `fetch` and `send` operations.

Fortran-Plus, Fortran-90, DAPL, and Parallaxis do not provide facilities analogous to `send` and `fetch`. The topologies that can be constructed in DAPL and Parallaxis must by definition be static (i.e. specified at compile-time) and cannot depend on values computed at run-time. CM-Lisp [Thinki87] provides an abstraction of global communication called a *xapping*. A *xapping* is a topology specified in terms of the *contents* of data-elements: elements with the same value are "connected" together. This permits interconnections of arbitrary fan-in and fan-out to be specified. The Paralation model's *mapping* is a similar concept. Communication is performed by the use of a *move* (`->`) operation. The Paralation model does not distinguish between local and global communication and does not provide separate constructs to express local communication.

The four operations: `imap`, `rotate`, `send`, and `fetch` are sufficient for data-parallel programming. The remaining primitives provide facilities for creating new arrays (`newarray`), finding the index of the last element (`bound`), selecting a single element (`select`), and pairing-up elements of conformant arrays (`zip`):

`newarray` : Int → α → array(Int, α)

`newarray` $n\ v \Rightarrow \ll v_0, \ldots, v_n \gg$

`bound` : array(Int, α) → Int

`bound` $\ll x_0, \ldots, x_n \gg \Rightarrow n$

`select` : Int → array(Int, α) → α

`select` $i \ll x_0, \ldots, x_n \gg \Rightarrow x_i$

`zip` : array(Int, α) → array(Int, β) → array(Int, α × β)

`zip` $\ll x_0, \ldots, x_n \gg \ll y_0, \ldots, y_n \gg \Rightarrow \ll (x_0, y_0), \ldots, (x_n, y_n) \gg$

The `imap` operation makes the programmer aware of strict locality restrictions because the function supplied as an argument to `imap` can only operate on individual array elements: it cannot refer to elements at other locations in the array. Where a function is required to operate on multiple elements residing in different locations in an array, they must first be moved into place by the use of the communication primitives and then paired up by `zip`. This strict enforcement of locality can be contrasted with the situation in imperative languages where any variable may appear on the right-hand side of a program statement and the programmer remains unaware of the cost of disregarding locality.

A data-parallel operation applies to every element of an aggregate, independently of all other elements in the aggregate. Therefore, data-parallel operations have $O(x)$ complexity where x is the complexity of the operation being applied. In the case of the first-order operations above, the complexity of all operations (except `fetch` and `send`) is O(1). The complexity of `fetch` and `send` is determined by the pattern of communication: if every element is routed to a different destination, it is potentially O(1), whereas if all elements are routed to the same destination, it is linear in the number of elements. If every element has a dedicated communication link to every other, then the interconnection pattern is termed a *full crossbar*. Such an interconnection facility is rare due to the number of links required (n^2). Contention arises whenever links are shared and the degree of contention that occurs is a function of the interconnection network and the network traffic. Contention on a link is generally resolved by queueing communication requests in some way. Therefore, if all messages are routed to a single destination, no communication can take place in parallel and the contention is resolved by sending all messages one after the other (i.e. sequentially).

An exception occurs in the case of machines with so-called *combining networks* (e.g. the Connection Machine) where messages bound for the same destination may be combined (reduced) *en-route*. Values are sent to intermediate processing elements where a function is applied to all messages headed for the same location. The functionality of such a network is captured in our scheme by the derived operation, `scatter`, defined in the next section. The complexity of an operation taking advantage of a combining network is $O(log(n))$ instead of $O(n)$ (in the worst-case) in the non-combining network. To take advantage of the capabilities of a combining network in some physical architecture, it is possible to redefine `scatter` as a primitive operation. The clear separation between operations performing computation and communication is then lost, however.

2.5. Derived Operators

The following operations (some of them commonly encountered in the literature) are defined in terms of the built-in operations defined in the previous section.

2.5.1. `map`, `map2`

The conventional form of `map` is defined as follows:

```
map : (α → β) → array(Int, α) → array(Int, β)

map = λf.λar. imap (λj.λx. f x) ar
```

A map for dyadic functions (this can be generalized to n-ary functions) is:

```
map2 : (α → β → γ) → array(Int, α) → array(Int, β) →
        array(Int, γ)

map2 = λf.λar.λbr. map (λ(x, y). f x y) (zip ar br)
```

Fortran-Plus and Fortran-90 provide access to data-parallel computation by overloading the basic arithmetic/logical operations on arrays. The statement `A = B+C` (where `A`, `B`, and `C` are arrays) uses a monomorphic form of addition to add all elements of `A` and `B` in parallel. Conditional assignment is performed by the addition of a parallel selection mode on arrays. For example:

```
A = 1
A(B .GT. C) = 0
```

the first statement sets all locations in `A` to `1` and the second uses the parallel selection mode to conditionally assign all elements of `A` where the corresponding element in `B` is greater than the corresponding element in `C` to `0`. Alternatively, the `MERGE` statement can be used:

```
A = MERGE(B, C, M)
```

Which assigns to `A` all elements of `B` where the corresponding location in the array of booleans, `M`, is true. Where `M` is false, the corresponding elements in `C` are assigned to `A`. The absence of any parallel form of conditional statement makes hiearchical conditional statements difficult: the programmer is required to manipulate these *activity masks* explicitly and use a programming style which relies on conditional side-effects. This leads to awkward code which is prone to errors and difficult to understand.

Parallaxis clearly separates parallel from sequential code by the introduction of a block-structuring statement, `PARALLEL-ENDPARALLEL` which permits parallel forms of conditional statements to occur within the block:

```
PARALLEL
  <parallel-statements>
  IF <vector-exp> THEN <parallel-statements> END
  <parallel-statements>
ENDPARALLEL
```

Alternatively, a subset of elements, based on their index, can be selected by supplying a range of elements to the `PARALLEL` statement:

```
PARALLEL [20..40]
   <parallel-statements>
ENDPARALLEL
```

The advantage of `imap` is that both of these selection mechanisms are available within a single operation and there is no need to perform conditional forms of assignment. The Fortran-Plus example using conditional assignment can be expressed as follows:

```
map2 (λ(b, c). (> b c) → 0; 1) B C
```

The function argument takes a pair of values and depending upon whether `b` is greater than `c`, returns either `0` or `1` for a particular element in the resulting array. Conditional statements may be nested arbitrarily. Subsets of an array can be selected by using `imap` since the index is available to the function argument as a parameter.

2.5.2. update

The `update` operation takes an index, an element, an array, and returns a new array with the value of the element at the index.

```
update : Int → α → array(Int, α) → array(Int, α)

update = λi.λv.λar. imap (λj.λw. (= i j) → v; w) ar
```

2.5.3. permute

The `permute` operation can implement any one-to-one and onto projection on arrays.

```
permute : array(Int, Int) → array(Int, α) → array(Int, α)

permute = λdr.λar. map hd (send dr ar)
```

2.5.4. shift

`shift` is similar to `rotate` except that edge-elements do not "wrap" around, instead some new value is assigned to locations that have been vacated. Both Fortran-Plus and Fortran-90 provide separate primitives to distinguish between `rotate` and `shift`. Using the primitives in this chapter it is possible to define `shift` in terms of `rotate`.

```
shift : Int → α → array(Int, α) → array(Int, α)

shift = λi.λv.λar.
   rotate i
      (imap (λj.λx. (≤ 0 (+ i j)) ∧ (≤ (+ i j) (bound ar)) →
            x; v) ar)
```

2.5.5. scan

`scan` is also sometimes called *parallel-prefix*. The `scan` operation applies an associative function to all initial segments of an array. For example, `scan (+) ≪1, 2, 3, 4≫` yields the array `≪1, 3, 6, 10≫` (i.e. the sum of [1], [1, 2], [1, 2, 3], and [1, 2, 3, 4]). The `scan` operation is surprisingly useful in a wide range of parallel algorithms. Blelloch proposes that `scan` should be considered a primitive parallel operation and details its usefulness in formulations of quicksort, a minimum spanning-tree algorithm, a merging algorithm, and various other applications [Blello89]. Fortran-Plus and Fortran 90 do not provide a `scan` operation.

The `scan` operation can be implemented in parallel by taking an array and rotating it by increasing powers of 2. After each rotate, the associative function is applied to all pairs of elements from the original and rotated array to produce a new array (which is used in the next rotation):

```
scan : (α → α → α) → array(Int, α) → array(Int, α)

spin : (Int → Int → α × α → α) × Int × Int × array(Int, α) →
       (Int → Int → α × α → α) × Int × Int × array(Int, α)

scan = λf.λar.
   let n = ceil (lg (+ 1 (bound ar))) in
   let (_, _, _, rr) = (spin ^ n)
      (λk.λj.λ(x, y). (> j k → f x y; x), 1, 0, ar) in
        rr

spin = λ(f, i, k, ar). (f, (* i 2), (+ k i),
   imap (λj.λx. f k j x) (zip ar (rotate i ar))
```

The `spin` function performs the rotation and the pairing-up of the original and rotated array. In the definition of `scan`, the underscore ('_') is used to denote unwanted elements of a returned tuple. The `lg` function computes the base-2 logarithm. The `scan` operation has a parallel complexity measure asymptotic to $log(n)$ where n is the number of elements (as can readily be deduced by inspection).

2.5.6. reduce

The `reduce` operation is equivalent to performing a `scan` and selecting the last element in the resulting array.

```
reduce : (α → α → α) → array(Int, α) → α

reduce = λf.λar. select (bound ar) (scan f ar)
```

The reduction operation is called *insert* (/) in FP. Instead of the general-purpose definition of `reduce` provided here, Fortran 90 provides just 7 pre-defined reduction operations: `SUM`, `PRODUCT`, `MAXVAL`, `MINVAL`, `COUNT`, `ANY`, `ALL` and `ALL`. The Parallaxis model provides a more general reduction operation, `REDUCE`.*f* (defined on all topologies) where *f* can be any user-defined function. Similarly, CM-Lisp and the Paralation model provide a reduction facility in the form of an additional function argument to a data-movement operation (the function is used to reduce elements arriving at the same destination).

2.5.7. scatter

`scatter` is a generalization of `reduce`. Elements arriving at the same destination are reduced by applying the function supplied as an argument.

```
scatter : (α → α → α) → α → array(Int, Int) → array(Int, α) →
          array(Int, α)
fold : (α → β → β) → β → list α → β

scatter = λf.λe.λdr.λar. map (fold f e) (send dr ar)

fold = λf.λa.λvs. (= vs []) → a; f (hd vs) (fold f a (tl vs))
```

Each element of `ar` is routed towards a destination specified by the corresponding index in `dr`. This operation uses the `send` operation and is therefore dependent upon the operational behaviour of the global routing interconnection mechanism. If the interconnection is non-deterministic (i.e. the order in which messages arrive cannot be guaranteed), the function `f` must be commutative in order to yield a deterministic result.

2.5.8. `gather`

`gather` is analogous to `scatter` except the array `dr` specifies the index of elements that are to be fetched, rather than sent (from `ar`), to the current location.

```
gather : (α → α → α) → α → array(Int, Int) → array(Int, α) →
         array(Int, α)

gather = λf.λe.λdr.λar. map2
   (λx.λys. (= ys nil) → f x e; f x (hd ys)) ar (fetch dr ar)
```

2.6. Alternative Aggregates

Arrays are not the only kinds of aggregate data-structures which can be used for data-parallel programming but this thesis restricts itself mainly to arrays. Some alternative aggregates can be supported in terms of an underlying array implementation. The remainder of this chapter presents an outline of possible alternative aggregates and monolithic operations which can be defined on them.

2.6.1. Classification of Aggregate Data-Structures

Aggregates can be classified by how their elements are accessed. Three classifications of aggregates can be identified: those that are accessed by location, content, or structure. These classifications correspond to arrays, bags or sets, and algebraic data-types (e.g. lists, trees, etc.) respectively. Array elements are accessed by their index, corresponding to their location in the array. Bags or sets are content-addressable data-structures where the elements are accessed by their value (e.g. $\{x \mid x < 3, x \in X\}$). Aggregates of algebraic data-types are accessed via their constructors (e.g. `cons`, `leaf`, `node`, etc.) which correspond to the structure of a data-object.

2.6.2. Exploiting Data-Parallelism with Alternative Aggregates

Data-parallel programming with content- and structure-addressable aggregates is presented next. The case of content-addressable aggregates is particularly interesting because of hardware and software technologies which have been developed to exploit parallelism in content- matching.

2.6.3. Content-Addressable Aggregates: Bags & Sets

Bags or sets are rarely used in conventional programming languages despite the usefulness of content-addressable aggregates because the overheads involved (in sequential implementations) rarely justify their use. Alternatives like hashing techniques or indexed-lookups on arrays are therefore preferred. In contrast, associative matching of elements with specialized hardware can be effective [O85]. General content-addressable access is the only feasible scheme when it is impossible to anticipate which data elements will be required. This is the motivation that led to the development of the content-addressable file store (CAFS) [Carmic85].

The availability of parallel architectures makes content-addressable data-structures not only feasible but particularly effective (especially on SIMD architectures). The Linda system relies on the use of a heterogeneous aggregate called *tuple-space* which acts as a data-flow synchronization medium between interacting sequential processes [Leicht89]. Data is deposited in tuple-space by processes where it can be retrieved by other processes matching on (some) of the contents of the data [Gelern90].

Potential applications which stand to benefit from efficient techniques for general content-matching include the comparison of protein sequences [Collin88] and text retrieval. Bags are a convenient aggregate data-structure for providing this form of functionality at the language level. Bags can be implemented as arrays (sets can then easily be implemented via bags). Operations on bags consist of either matching operations on the value of elements in a bag, or some form of reduction operation defined on bags. Selection operations can be translated into a `map` operation on arrays.

For example, consider a content-matching operation on a bag B such as: $\{x \mid (x, 3) \in B\}$. This is equivalent to the following program defined on arrays:

```
getx : array(Int, (α × Int)) → array(Int, α)

getx = λbr. map (λ(x, i). (= i 3) → x; void)
```

A special value is required, which has every type, called `void`. This value can be returned for all locations which are not part of the bag represented by the array (i.e. `void` elements correspond to "empty space" in the bag). All primitive operations need to be overloaded on the `void` value so that if it is supplied as an argument of any primitive

dyadic operation, the operation returns the other argument. In this way, reduction on bags can simply use the existing `reduce` function defined on arrays.

Brute-force matching algorithms are particularly effective on SIMD architectures where thousands of processing elements are available to perform the comparisons in parallel. Content-addressable data-structures are therefore well suited for the exploitation of data-parallelism.

2.6.4. Structure-Addressable Aggregates: Lists, Trees, etc.

Functional languages usually employ a *pattern-matching* mechanism which allows constructors in the left-hand side of an equation to act as destructors on algebraic data-types. For example, in the program below:

```
sumtree : tree(α) → Int

sumtree = λempty. 0
        | λleaf a. a
        | λtree lt rt. (+ (sumtree lt) (sumtree rt))
```

This program uses pattern-matching on the structure of the argument (a tree) to cause the appropriate equation to be selected. Algebraic data-types are therefore accessed by structure. It is possible to embed an arbitrary algebraic data-type in an underlying array implementation by randomly allocating elements tagged with their constructors to locations in the array. Pointers to other components of an algebraic data structure are represented by indices of the array elements where the components are located. Figure 2.2 shows a possible embedding of trees in an array. This approach can be generalized to any arbitrary algebraic data-type. It is possible to define versions of `reduce` and `map` which can obtain data-parallelism from algebraic data-types embedded in arrays.

Embedding Tree Data-types into Arrays

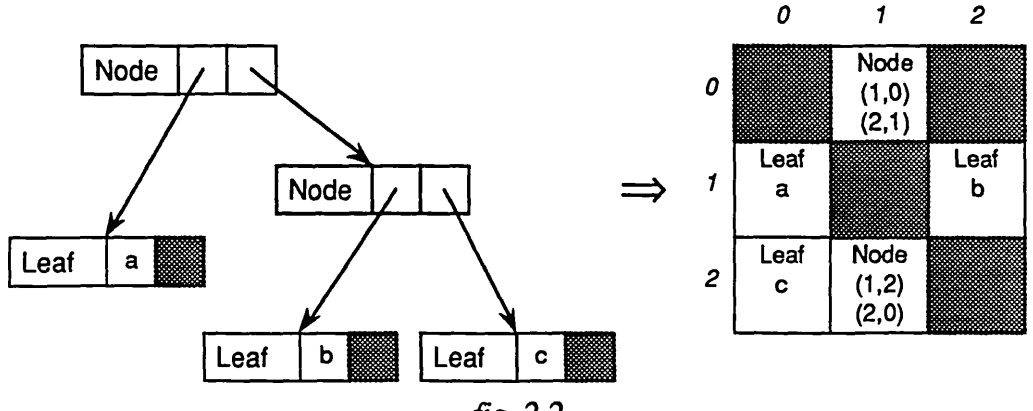


fig. 2.2

Although the embedding of algebraic data-types in arrays is feasible, to exploit data-parallelism it is necessary to supply monolithic operations as primitives because instantaneous access to all elements of an aggregate is required.

2.6.5. Operators for Alternative Aggregates

Some of the operations (e.g. `map`, `reduce`, etc.) are *structure-invariant* in the sense that it is possible to induce analogous definitions on other types of aggregate structures for these operators using the so-called Boom hierarchy (see [Backho89] for an elegant exposition of this theory) for bags and sets.

The theory relies on recursive definitions for aggregate data-structures and imposing associativity, commutativity, and idempotency (ACI) constraints on the *join* operation which takes two aggregates and combines their elements into one, yields the semantics of trees, arrays/lists, bags, and sets respectively.

The ACI hierarchy is shown in fig. 2.3. The operations given previously can be defined in terms of *homomorphisms* which embody the algorithmic skeleton of a structure-invariant operation. Homomorphisms are defined on recursively defined aggregates and therefore inherit properties of the *join* operation (used to construct data structures from atomic components). Therefore, `reduce`, defined as a homomorphism on bags, inherits the associativity and commutativity constraints on the join operation defined on bags. The function argument performing the reduction must therefore be associative and commutative in order to yield a deterministic result.

ACI Hierarchy for Join Operation on Aggregates

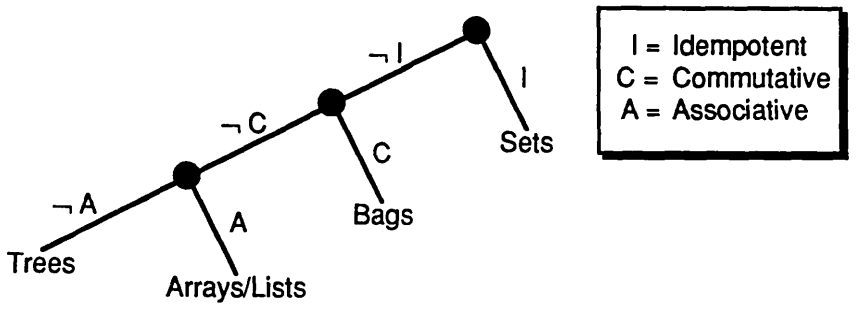


fig. 2.3

The associativity of a function cannot, in general, be determined using static (compile-time) or dynamic (run-time) checks. To prove that a binary function $\oplus$ on elements from a set T is associative, it is necessary to show that the triple $(\oplus, e, T)$ forms an Abelian group (where e is the identity element of $\oplus$). This is undecidable for non-finite sets T and unwieldy for finite T. It must be established during program specification and refinement by producing rigourous logical proofs that this is the case. Compile time and run-time checks can help in discovering some incorrect uses of functions (e.g. associative functions must have type $\alpha \times \alpha \rightarrow \alpha$) but the general problem is beyond the ability of existing program analysis methods. This illustrates a problem with the exclusive use of executable notations for programs.

The theory behind this work is beyond the scope of this thesis and is introduced only as a means for describing how the theory behind the operations developed in this chapter can be extended to incorporate monolithic operations on other aggregate data-types. The interested reader is referred to [Bird87], [Malcol89], [Marino89], and [Spivey89]. Given that other aggregate data-types can be embedded in arrays, it is possible to continue with arrays as the primary aggregate data-type without any loss of generality.

Chapter 3

3. Data-Parallel Functional Programming

With a functional language extended by a set of data-parallel operations, it is possible to develop programs exhibiting data-parallelism in the functional style. Such programs retain the traditional benefits accrued to functional languages in terms of clean semantics, high expressiveness, rapid prototyping, and ease of maintenance with the additional benefit of exhibiting massive parallelism on data-parallel architectures.

Using the data-parallel operations defined in the previous chapter, a set of data-parallel programs are demonstrated which illustrate the power and flexibility of these operations. The chosen sample programs reflect a desire to adopt application domains which traditionally have not been successful in a functional setting because of the poor performance of such algorithms in existing implementations. Furthermore, some of the latter applications in this chapter have been chosen because they are unusual and it is not immediately obvious how data-parallelism can be brought to bear on their solution.

3.1. Sample Applications

The sample applications in this chapter have been chosen to demonstrate a functional style of data-parallel programming. Programming with monolithic, higher-order operations is consistent with existing programming practices in functional languages so the extensions proposed in the previous chapter are easily assimilated because they are a natural "fit." In particular, the kind of applications in this chapter are from an application domain which has not traditionally attracted the attention of functional programmers because implementations of these types of programs has traditionally been very inefficient on scalar or MIMD architectures. The presentation of these programs has two aims:

- To demonstrate the way in which the data-parallel operations presented in the previous chapter can be used to write data-parallel functional programs.

- To attempt to persuade the reader that such programs are naturally expressive, elegant, and efficient formulations of algorithms.

The programs in question include the evaluation of polynomials in parallel, a solution to the histogram problem, a parallel version of bubble-sort, the simulation of digital logic circuits, Gaussian elimination, and an algorithm for displaying images compressed using fractal-encoding methods.

3.1.1. Polynomial Evaluation

The first example consists of evaluating a set of polynomials at a variety of points. The evaluation of polynomials is often used in computing keys for hashing algorithms and for interpolation. The goal of this example is to illustrate the following points:

- Existing user-defined functions can be used to exploit data-parallelism by using them as arguments to `map`.

- Such functions may be recursive (i.e. the length of the execution thread at each PE may vary).

- Arrays may contain elements which are algebraic data-types (in the current example, lists).

The value of a polynomial of degree n, P_n at some value x is given by:

$$P_n(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x_1 + a_0$$

A polynomial is completely described by the values of its coefficients (a_n, ... a_0). A linear-time algorithm to calculate the value of a polynomial known as *Horner's Rule* is commonly used. Horner's Rule is defined as:

$$P_n(x) = \sum_{i=0}^{n} a_i x^i = (\ldots((a_n x + a_{n-1})x + a_{n-2})x + \ldots)x + a_0$$

If the coefficients of a polynomial are stored in a list (`as`) in reverse order: [a_0, ..., a_n], then the following function uses Horner's Rule to return the value of the polynomial at some value of `x`:

```
poly = λx.λas. fold (λa.λc. (+ (* c x) a)) 0 as
```

The `fold` function was defined in the previous chapter. `poly` can be used in data-parallel fashion by mapping it to an array whose elements are lists of coefficients to compute the value of polynomials at a number of distinct points in parallel. The function `polyarray` therefore takes a single `x`, and maps the partially-applied function `(poly x)` to each location in the array `ar`:

```
polyarray : Real × array(Int, list Real) → array(Int, Real)
polyarray = λx.λar. map (poly x) ar
```

Where `ar` is an array of lists of coefficients. The novelty rests in that the array can contain the coefficients of polynomials of *varying degree.* In other words, the length of the lists at each element in the array may vary. The compilation scheme in Chapter 5 ensures that execution of `poly` continues at each location for the length of the local list. The running time of the program is proportional to the length of the longest list in the array.

3.1.2. Histogram

This example is the histogram problem (posed as a challenge to find an efficient functional realization in [Arvind86]). Given an array of values of ordinal type, produce a count of how many times each value occurs in the source array. The classic application is to produce a count of the number of times a character occurs in a string of text (represented as an array whose elements are of type *Char*):

```
hist : array(Int, Char) → array(Int, Int)
hist = λar. scatter (+) 0 (map ord ar) (map (λx. 1) ar)
```

The original array of characters is converted into an array all of whose elements are set to `1`. These new values of the array elements are routed to a location specified by the ASCII value of the character originally at each location (`ord` converts a character to its

ASCII value). All the 1s arriving at the same location are summed by the + operation. The result is that each of the first 128 elements in the resulting array contains the sum of the number of times each particular character (whose ASCII value corresponds to the index) occurred in the original text. The routing pattern of data-elements is determined only at run-time (i.e. it is dynamic), which requires the `send` operation used in the definition of `scatter`. In data-parallel languages which are restricted to static topologies (e.g. DAPL, Parallaxis), the solution to the histogram problem becomes much more difficult to specify.

3.1.3. Parallel Bubble-Sort

On scalar architectures, the traditional bubble-sort method is terribly inefficient with complexity $O(n^2)$. On a SIMD machine, however, it is possible to implement a parallel version of this sorting algorithm which has the desirable property that all inter-PE communication is nearest-neighbor. Faster parallel sorting algorithms exist, but this sample program illustrates the use of nearest-neighbor inter-element locality exploited by `rotate`.

The algorithm works as follows: The elements to be sorted are in a one-dimensional array. First, all elements with index i, where i is an even number, compare themselves with the value at location $i+1$. If the neighboring value is smaller, they swap locations. Otherwise, the values stay where they are. Secondly, all elements at odd-valued indices effect a swap. If this is repeated $\lceil n/2 \rceil$ times (where n is the number of elements in the array, then the elements will be sorted.

In contrast with the previous two examples, in this exercise the solution will be shown in the language used to program the DAP, Fortran-Plus, followed by the solution expressed in our functional language. In this way the difference in the programming styles can be directly compared for a specific application. The Fortran-Plus example is given first:

```
C
C  B U B B L E _ S O R T :
C
       SUBROUTINE BUBBLE_SORT(AR)
       INTEGER*4 AR(*)

       CALL BUBBLE_SORT2(AR, SIZE(AR))

       RETURN
       END
C
C  B U B B L E _ S O R T 2 :
C
       SUBROUTINE BUBBLE_SORT2(AR, N)
       INTEGER*4 AR(*)
       INTEGER*4 N

       LOGICAL M1(*SIZE(AR)), M2(*SIZE(AR))
```

```
      CALL MAKE_MASKS(M1, M2, AR, N)

      DO 10 i = 1, (FIX(0.5 + (N+1)/2))

      CALL EXCHANGE(M1, M2, AR, N)
      CALL EXCHANGE(M2, M1, AR, N)

10    CONTINUE

      RETURN
      END
C
C  M A K E _ M A S K S :
C
      SUBROUTINE MAKE_MASKS(M1, M2, AR, N)
      LOGICAL M1(*SIZE(AR)), M2(*SIZE(AR))
      INTEGER*4 AR(*)
      INTEGER*4 N

      M1 = .NOT. ALT(1, N)
      M2 = .NOT. M1

      RETURN
      END
C
C  E X C H A N G E :
C
      SUBROUTINE EXCHANGE(M1, M2, AR, N)
      LOGICAL M1(*SIZE(AR)), M2(*SIZE(AR))
      INTEGER*4 AR(*)
      INTEGER*4 N

      LOGICAL M3(*SIZE(AR))
      INTEGER*4 BR(*SIZE(AR)), XO(*SIZE(AR)), XE(*SIZE(AR))

      BR = SHRC(AR)
      BR(1) = MIN
      M3 = (BR .GT. AR) .AND. M2
      XO = MERGE(BR, AR, M3)

      BR = SHLC(AR)
      BR(N) = MAX
      M3 = (BR .LT. AR) .AND. M1
      XE = MERGE(BR, AR, M3)

      AR = MERGE(XE, XO, M1)

      RETURN
      END
```

Fortran-Plus relies on overloading basic arithmetic/logical operations on arrays to provide access to data-parallel computation. These overloaded operations have been shown in boldface. The ALT primitive is used to construct an array of alternating true/false values which corresponds to the mask of array elements whose index is even. The negation of this mask is the mask corresponding to all odd elements. These two masks are assigned to two array variables, M1, M2 to be used at later points in the program. The main part of the program is performed in the EXCHANGE subroutine. The swapping of neighboring elements

is performed by shifting the original array to the right, selecting the maximum values, then shifting the original array to the left, and selecting the minimum values.

The original array AR is shifted to the right by SHRC, the vacated first element is set to a minimum sentinel value, denoted by MIN. The absence of a parallel form of conditional statement means that it is necessary to construct a mask corresponding to all even-index locations where the shifted array contains a larger value than the unshifted array. This is stored in M3. The maximum values from the shifted and unshifted arrays are merged into XO. The same procedure is performed to select all the minimum values, using a maximal sentinel value, MAX. The result is assigned to XE. The final result consists of the appropriate elements from both XO and XE, merged together by the use of the appropriate mask. The subroutine BUBBLE_SORT2 calls EXCHANGE twice. The first time to perform a swap at all even index locations, the second time to swap all odd-index locations. The elements at the appropriate indices can be swapped simply by exchanging the mask parameters, M1 and M2.

The equivalent functional program is given below:

```
bubsort = λar.
   ((swap odd ∘ swap even) ^ (ceil (/ 2 (+ 1 (bound ar))))) ar;
```

where:

```
swap = λp.λar.
   let flip = λj.λ(a, (r, l)). p j → (lt a  l); (gt a r) in
       imap flip (zip ar (zip (shift 1 MIN ar)
                              (shift -1 MAX ar)))

even = λj. (= (mod j 2) 0)

odd = λj. (= (mod j 2) 1)

gt = λx.λy. (> x y) → x; y

lt = λx.λy. (< x y) → x; y
```

The resulting program is much more co[illegible]e and understandable. The absence of parallel conditionals in Fortran-Plus require[illegible]e programmer to manipulate masks explicitly. In the functional program this is unnecessary. The Fortran-Plus programmer also has to explicitly combine partial results (e.g. XO and XE) to yield a final result. This leads to a programming style which relies on side-effects (assignment). In the functional program, the results returned by both branches of a conditional statement are automatically merged into a single result array. The resulting program requires $O(n)$ PEs and has performance asymptotic to $O(n)$.

3.1.4. Digital Logic Circuit Simulation

A natural affinity exists between functional languages and digital logic devices in the sense that simple logic gates can be modelled as functions and programs can be written as compositions of these functions to simulate more complex logic devices. Simulating the behaviour of complex logic circuits is invaluable in the design of digital controllers, ASICs (Application-Specific Integrated Circuits), and programmable gate-arrays. In these devices, complex hierarchies can result involving feedback loops and multiple stages of logic gates. It is important to be able to verify designs prior to fabrication.

Simulation of logic circuits via functional languages is not new. A design methodology based on expressing logic circuits in the functional language FP is described in [Patel85]. Programs to simulate the behaviour of digital circuits exhibit a great deal of inherent parallelism. The fine-grained nature of the computation involved suggests an efficient implementation might be possible on data-parallel architectures whereas the complex interconnection possibilities and hierarchies seem to preclude a good data-parallel implementation. Nevertheless, as will be demonstrated, such applications can readily be programmed in our extended functional language to yield highly data-parallel programs.

This application illustrates the following points:

- Arbitrary cycles can be modelled by the use of communication primitives.

- User-defined data-types can be useful in modelling heterogeneous arrays (i.e. arrays whose elements are functionally distinct).

- Lazy-evaluation lends itself well to the expression of inherently "infinite" (i.e. non-terminating) applications (such as simulation).

For example, consider the simple circuit in fig. 3.1. There are 3 inputs to this system, labelled *a*, *b*, and *c*. The gates marked 0, 1, 4, and 5 are AND gates whereas 2, 3, and 6 are OR gates. Conventional algorithms for circuit simulation represent the circuit as a graph where the gates form the nodes and the wires are the edges.

The absence of state and the presence of cycles makes such an approach difficult in a functional language. The potential difficulties with a data-parallel implementation of circuit-simulation are:

- The presence of hierarchies limit parallelism because of dependencies between levels in the hierarchy.

- The presence of cycles (how should they be represented?).

- Heterogeneous elements (different types of gates in the circuit) reduce parallelism because they introduce conditional-statements.

3-Input Logic Device

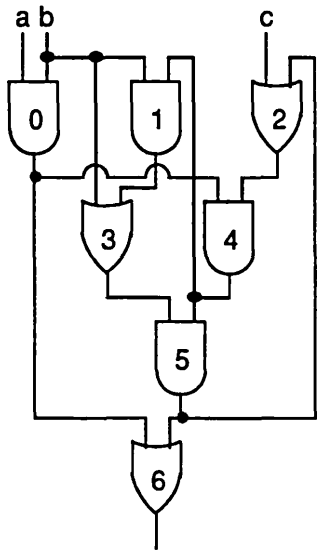


fig. 3.1

The approach taken in this paper relies in representing the edges of the graph as destinations used by the `send` primitive. The signals travelling along the wires in a digital circuit are voltages (low/high) corresponding to logical 0 and 1. This can be defined as an algebraic data-structure:

```
signal = 0 | 1 | X
```

An additional state, `X`, has been added which stands for "either `0` or `1`." The use for this additional state will be shown later. Without loss of generality, it is assumed that all logic gates have 2 inputs and produce a single output. The AND, OR gates can be defined as functions over the domain `signal`. The AND gate returns `1` if both of its inputs are `1`. If one of its inputs is `0`, then it returns `0`. If one input is `1` and the other is `X`, then it returns `X`. The OR gate returns `0` if both of its inputs are `0`. If one of its inputs is `1`, then it returns `1`. If one input is `0` and the other is `X` then it returns `X`. Both gates return `X` if both inputs are `X`.

The gates in the circuit are presented linearly as in fig. 3.2. The input and output of each gate is assigned a unique label. From the diagram a "wiring list" is produced which describes the source of the inputs of each gate in the circuit. An array is defined where each element corresponds to the output of the gates (0-6), extended with the inputs (*a*, *b*, *c*). The input of each gate can now be expressed as the index of a location in this array. Since each gate has two inputs, two arrays are required (see fig. 3.3). For example, the inputs of

gate 2 come from *c* and the output of gate 5. The value of the input *c* is stored in location 9 and the output of gate 5 at location 5. The locations corresponding to the inputs of *a*, *b*, and *c* have been "greyed-out" to indicate that they do not require inputs from anywhere.

Another array of the same size as the arrays in fig. 3.3 is created where the elements are set to the initial values of the outputs of each gate. Initially, the outputs of all the gates are unknown and this is indicated by assigning them the value `x`. By introducing this "either `1` or `0`" state, it is possible to track the propagation of valid signals through the circuit. Assume that the inputs *a*, *b*, and *c* are set to `0`, `1`, and `1` respectively.

Each gate can now fetch its inputs from the locations provided in fig. 3.3. During each cycle, the corresponding inputs are fetched for each gate, the appropriate function (AND, OR, etc.) is applied to the two inputs at each location, yielding a new output for

Linearized Gates (2-input/1-Output)

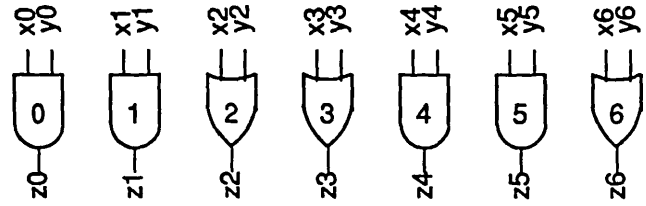


fig. 3.2

Gate Inputs

0	1	2	3	4	5	6	7	8	9
7	8	9	8	0	3	0			
8	4	5	1	2	4	5			
z0	z1	z2	z3	z4	z5	z6	a	b	c

fig. 3.3

each gate. In successive cycles, input values propagate through the circuit, changing `x` outputs into 0 or 1. For the circuit in fig. 3.1, the array of output values of each gate after each cycle is shown in fig. 3.4.

The output of the program is a list of arrays corresponding to the outputs of the gates in fig. 3.4. Alternatively, this result can be displayed as a standard timing diagram (as in fig. 3.5). How are the different functions applied to the inputs? An array is created where the elements are set to constructors which can be discriminated against. For example, the type `gate` defined below:

```
gate = andgate | orgate | fst
```

Suggests an array: `array(Int, gate)`, where elements 0, 1, 4, and 5 are set to `andgate` and 2, 3, and 6 to `orgate`. These constructors are used in a `case`-statement so

that the appropriate function (AND, OR) can be applied to the arguments at the appropriate locations. An additional constructor, `fst`, is introduced to handle the locations corresponding to the inputs *a*, *b*, and *c*. These elements in the array do not correspond to any logic gate. The inputs for each of these locations is the current index (i.e. 7 for *a*, 8 for *b*, and 9 for *c*) so that *a*, *b*, and *c* in effect "cycle round" and re-appear at the same location during each iteration (this corresponds to keeping the inputs steady during simulation). There are two inputs but both contain the same value so either can be selected. `fst` is defined to select the first value. This ensures that these locations will continue to contain

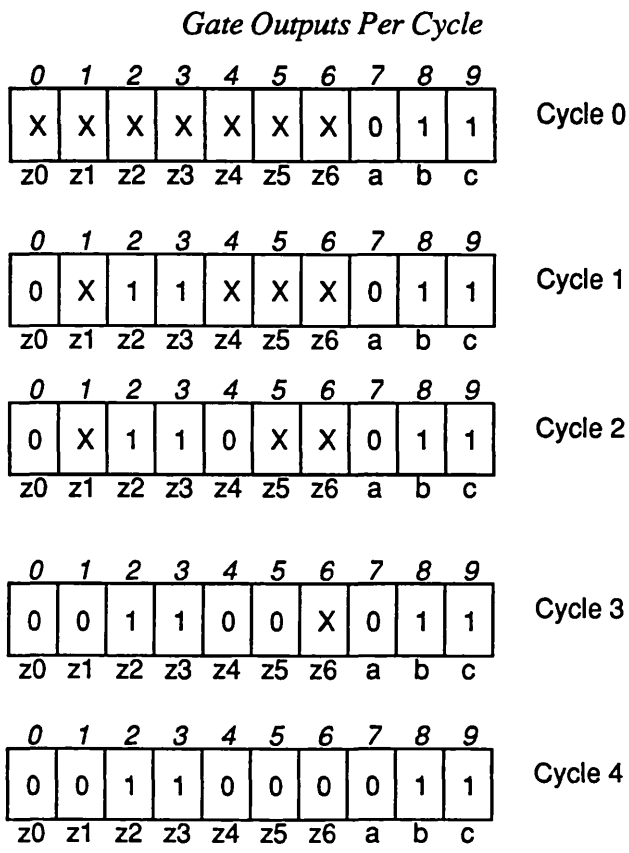


fig. 3.4

the original input values. The `act` function which applies a corresponding function to the locations in the array depending on the type of `gate` is (`FST = λx.λy. x`):

```
act = λg.λ(x, y).
   case g in
      when andgate : AND x y
      when orgate : OR x y
      when fst : FST x y
```

This function needs to be mapped to the values fetched according to the input arrays given in fig. 3.3 (labelled `in1` and `in2` below). The `cycle` function fetches both inputs for each gate and then applies `act`:

```
cycle = λin1.λin2.λgr.λrr.
   let xr = map hd (fetch in1 rr) in
   let yr = map hd (fetch in2 rr) in
      map2 act gr (zip xr yr)
```

Finally, a function, `generate`, applies `cycle` iteratively, producing an infinite list of output arrays corresponding to the output of each gate after each cycle. A suitable finite initial segment of this list can be evaluated to yield the desired result.

```
generate = λin1.λin2.λgr.λrr.
   let sr = cycle in1 in2 gr rr  in
      cons sr (generate in1 in2 gr sr)
```

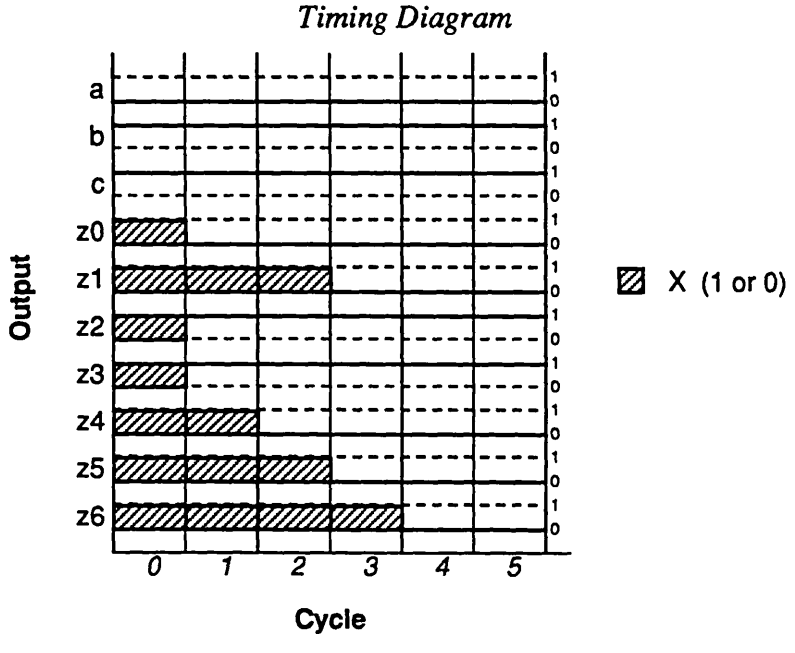


fig. 3.5

The higher-order data-parallel operations defined in the previous chapter permit a very concise specification of this application. Use is made of the facility provided by algebraic data-types to declare a heterogeneous array, containing elements corresponding to different logic gates. The techniques used to circumvent problems with cyclic data-structures in functional languages by representing graphs as arrays of nodes and edges can be generalized to a variety of common graph-algorithms. The communication primitives presented in the previous chapter implement the concurrent communication of values along edges in a graph. This facility is also exploited in §3.1.6.

3.1.5. Gaussian Elimination

A common requirement in many numerical algorithms in the natural and applied sciences is for the solution of systems of linear equations. The simplest method of solving linear equations directly is via Gaussian elimination, where a set of linear equations, for example:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 &= b_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 &= b_2 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 &= b_3 \end{aligned}$$

Can be described as a matrix product $\mathbf{Ax} = \mathbf{b}$.

$$\underset{\mathbf{A}}{\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}} \cdot \underset{\mathbf{x}}{\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}} = \underset{\mathbf{b}}{\begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}}$$

The goal of Gaussian elimination is to transform the **A** matrix into an upper-triangular form (the lower triangle consisting of elements set to 0) so that it becomes possible to solve for x_3 directly and in turn x_2, and x_1. Gaussian elmination is a good application to consider because:

- Solutions to linear equations are a common requirement of many applications in the pure and applied-sciences.
- Dependencies between data-elements are complex making parallelism difficult to exploit.
- Parts of the matrix remain unchanged by the computation.

The conventional method of Gaussian elimination is described below. Elimination is performed on the matrix **A'** which consists of the elements of the **A** and **b** matrices:

$$\underset{\mathbf{A'}}{\begin{bmatrix} a_{11} & a_{12} & a_{13} & b_1 \\ a_{21} & a_{22} & a_{23} & b_2 \\ a_{31} & a_{32} & a_{33} & b_3 \end{bmatrix}}$$

Compute the multipliers of rows 2 and 3:

$$m_{21} = \frac{-a_{11}}{a_{21}}, \quad m_{31} = \frac{-a_{11}}{a_{31}}$$

Multiplying rows 2 and 3 by their respective multipliers and adding the results to the first row yields:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & b_1 \\ 0 & m_{21}a_{22}+a_{12} & m_{21}a_{23}+a_{13} & m_{21}b_2+b_1 \\ 0 & m_{31}a_{32}+a_{12} & m_{31}a_{33}+a_{13} & m_{31}b_3+b_1 \end{bmatrix}$$

Which can be re-written as:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & b_1 \\ 0 & a'_{22} & a'_{23} & b'_2 \\ 0 & a'_{32} & a'_{33} & b'_3 \end{bmatrix}$$

Where $a'_{ij} = m_{i1}a_{ij}+a_{1j}$. So far, the first column of the matrix has been simplified. Proceeding for the second column, a new multiplier is computed:

$$m_{32} = \frac{-a'_{22}}{a'_{32}}$$

Multiplying the elements of the third row with the multiplier and adding the result to the second row yields:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & b_1 \\ 0 & a_{22} & a_{23} & b_2 \\ 0 & 0 & m_{32}a'_{33}+a'_{23} & m_{32}b'_3+b'_2 \end{bmatrix}$$

$$\Downarrow$$

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & b_1 \\ 0 & a'_{22} & a'_{23} & b'_2 \\ 0 & 0 & a''_{33} & b''_3 \end{bmatrix}$$

No more columns remain to be eliminated at this point. To solve for the values of x_1, x_2, and x_3, it is necessary to perform back-substitution starting with:

$$a''_{33}x_3 = b''_3$$

This yields the value of x_3, which can then be substituted in the previous row to yield x_2 and so on. The program in this section performs Gaussian elimination only. A program to perform back-substitution is not provided.

It is not directly obvious how a problem such as Gaussian elimination can be specified using monolithic array operations (as opposed to specifying the solution at the element level, as is the case in most conventional languages). This is why it has been included here. The Gaussian elimination algorithm presented here is remiss in that it does not attempt to perform pivoting.

The difficulties in a data-parallel and monolithic formulation of Gaussian elimination are due to:

- Multipliers must be computed for all rows in parallel.
- Multipliers must be propagated across the columns for all rows in parallel.
- Each row must be replicated across the remaining rows in parallel.
- Previously computed rows and columns must remain unchanged.

Replicating a Value Across a 1-D Array

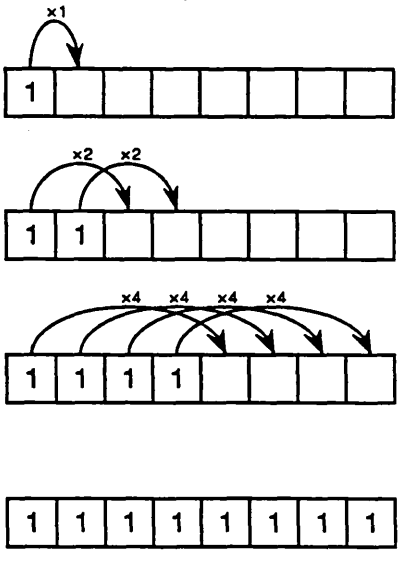


fig. 3.6

Two of these tasks involve exploiting parallelism in communication. This is explored in detail in the following example. A single value can be replicated across an entire 1-dimensional array in $lg(n)$ iterations,[1] where n is the number of elements in the array. To see how this can be done, see fig. 3.6. Replication involves a series of `shift` operations, each `shift` operation shifting elements twice as far as the previous one. This method can readily be applied to the two-dimensional case so that any column or row can be replicated across an entire array in $lg(n)$ steps (where n is the number of rows or columns).

[1]This assumes that a `rotate` operation takes constant time, regardless of the distance that elements are shifted, which is reasonably valid in architectures where `rotate` operations are pipelined (e.g. MasPar MP-1).

The strict enforcement of locality in our language ensures that the monolithic formulation of Gaussian elimination is primarily concerned with communicating values to the proper locations. For each iteration i, the i^{th} row must be replicated across the remaining rows so that the elements in row i can be added to the corresponding elements in the remaining rows, the multipliers must be computed, the multipliers must be replicated across all the remaining columns, and then multiplied with the original elements and the elements of the replicated i^{th} row. The function `elim`:

```
elim = λi.λn.λar.
   (= i n) → ar;
      let rr = replicate i S ar in
          elim (+ i 1) n (newval i ar (mults i ar rr) rr)
```

Data Movement Required in Gaussian Elimination

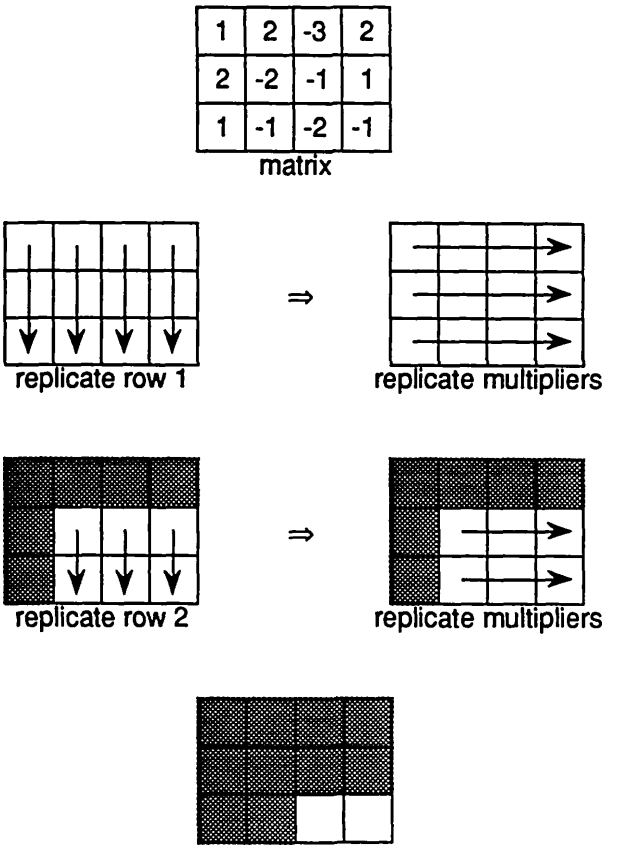


fig. 3.7

carries out n iterations of the elimination algorithm. The function `replicate` takes as arguments the row or column to be replicated (`i`), a direction (`N`, `E`, `W`, `S` corresponding to north, south, east, and west respectively), and the array `ar` (`replicate` can be defined readily in terms of the `spin` function in the previous chapter). The call to `replicate i S ar` yields an array where all rows greater than `i` contain a copy of the

elements in row `i`. The call to `mults` computes the multipliers and replicates them across the columns:

```
mults = λi.λar.λrr.
   replicate i E (imap
      (λ(jr,jc).λ(x,y). (= i jc) → (/ -y x); x) (zip ar rr))
```

The multipliers are computed in column `i` only. The conditional in the function argument to `imap` ensures that the values in the other columns remain unchanged. After the multipliers have been computed, they are replicated across the columns (to the east). The new value for each element in the rows and columns greater than `i` is then computed by `newval`:

```
newval = λi.λar.λmr.λrr.
   imap (λ(jr,jc).λ(a,(m,r)). (∧ (> jr i) (> jc i)) →
      (+ (* a m) r); a) (zip ar (zip mr rr))
```

Where `mr` is the array of replicated multipliers and `rr` is the array of replicated row `i`. The top-level function to perform Gaussian elimination on an array `ar` is `gauss`:

```
gauss = λar. let (jr,jc) = bound ar in elim 0 jr ar
```

The data-movement involved in replicating rows and columns for a system of 3 linear equations is show diagrammatically in fig. 3.7. The use of `imap` to selectively activate a portion of the array for the computation of new array values is shown by displaying inactive locations (i.e. those whose values are left unchanged) in grey.

The resulting program has parallel complexity of $O(n \cdot log(n))$ where n is the number of rows or columns in the matrix (two $log(n)$ replications are required for n iterations of `elim`). In this rendition of Gaussian elimination, the communication of data-elements is clearly separated from computation on those values. In recent years, there has been much interest in a class of data-parallel algorithms where communication and computation are interleaved to yield heterogeneous pipeline algorithms known as *systolic algorithms*. This particular subject is discussed in Chapter 7. It is possible to derive a systolic formulation of Gaussian elimination where communication and computation are interleaved and the distances for communication are reduced, yielding a more efficient algorithm.

The monolithic formulation of Gaussian elimination prompts the following observations:

- Strict enforcement of intra-element locality requires that algorithms be concerned with communication and how such communication can exploit parallelism.

- Elements of arrays whose values are to remain unchanged require explicit copying.

- Sequential formulations of Gaussian elimination on von-Neumann machines rely on a uniform, globally-accessible store and disregard the importance of locality.

- Functional programmers are accustomed to the lack of a global store so the communication of needed values is consistent with the functional style.

- The powerful primitive operations presented in the previous chapter make it possible to express the parallel forms of communication required in data-parallel programs.

3.1.6. Iterated Function Systems (IFS)

In January 1988, an article appeared in *Byte* magazine by Barnsley and Sloan in which they proposed a new method for the compression of images by fractal methods [Barnsl88]. Their system, which they termed *Iterated Function Systems* (IFS) involves specifying an image as a series of simple affine transformations. *Affine transformations* involve scaling, rotation and displacement. They can be specified as a mapping from one point (x, y) in a two-dimensional real coordinate space $(\Re \times \Re)$ to another (x', y'):

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} e \\ f \end{bmatrix} = \begin{bmatrix} x' \\ y' \end{bmatrix}$$

This new approach to compressing images provoked particular interest amongst the technical community because of the high compression ratios (10,000 to 1 or more) claimed for the technique and because of the elegant mathematical foundations of fractal geometry on which the scheme is based.

A compressed image is a series of affine transforms which together form an "attractor" which is the image. An efficient automatic algorithm for the compression of images to yield the affine transforms has not yet been found but the decompression of images is an area of active research. The original article describes an algorithm based on Monte-Carlo solution methods to decompress an image. Alternatively, a deterministic algorithm for decompression is described in [Monro90], based on starting from the fixpoints of the affine transforms. An alternative parallel algorithm is developed in [Sharp91]. A novel parallel algorithm is presented here by showing it to be isomorphic to

the breadth-first graph-traversal problem and then demonstrating a solution which exploits data-parallelism.

This particular application is demonstrated to illustrate the following points:

- A novel data-parallel implementation of an existing algorithm.

- This algorithm demonstrates a particular class of algorithms where the geometric decomposition of the problem, coupled with communication of messages yields a solution [Sharp91]. These algorithms can yield good data-parallel implementations.

- Illustrate how graph-manipulation algorithms (in this case, breadth-first traversal) can be implemented to exploit data-parallelism.

If an image is defined by 0 .. k affine transforms, then from a point (x, y) in the image, applying each transform to the point yields $k+1$ points which are also in the image. The decompression of an image therefore consists of starting from a point in the image and successively applying each of the transforms to the point and to all of its successors. In the real coordinate space ($\Re \times \Re$), there are an infinite number of points in the image. However, digital images are displayed using a finite number of discrete *pixels* (picture elements) so by projecting the image from real coordinate space ($\Re \times \Re$) to the integer coordinate space ($I \times I$), there are a finite number of points in the image and it becomes possible to derive a decompression algorithm which executes in bounded time. This translation, via a projection function $\wp$, is portrayed in fig. 3.8.

The remaining problem is to start from a point which is guaranteed to be in the image. The solution proposed by Monro was to compute the fixpoints of each of the transformations (i.e. the point (x, y) where $\tau(x, y) = (x, y)$ and τ is an affine transform). Finding the fixpoints of each transformation involves solving two simple linear equations.

Viewed another way, each pixel is a node in a graph and the transformations form edges to other pixels in the image (see fig. 3.9). The set of all pixels forms the graphical display, of which a sub-set forms the pixels in the image. The image M can therefore be described as the result of the following equation:

$$M = \bigcup_{i=0}^{k} (adj \circ fp)\, i$$

Where *adj* returns the set of all adjacent nodes from a point and *fp* returns the fixpoint of a transformation i. The result is a set of pixels, M, which form the image (see fig. 3.10) defined by 0 .. k transforms. In this implementation the set of nodes in a graph

is represented as an array and the edges connecting them are stored as 0 .. *k* arrays of destination indices in the array.

A breadth-first traversal of the graph will return the set of all adjacent pixels. This traversal consists of starting from the fixpoints of the image and then sending messages to each of the adjacent nodes, rendering them active. During each iteration, all active nodes send messages to their adjacent nodes. The state of each pixel in the array can be described via the following ADT:

```
state = off | on | lit
```

Which indicates that a pixel is either out of the image (`off`), has just been reached by the breadth-first traversal (`on`), or that it has been reached previously (`lit`). All pixels but the fixpoints are initially set to the `off` state. All pixels send messages to their

Projection from Real to Integer Space

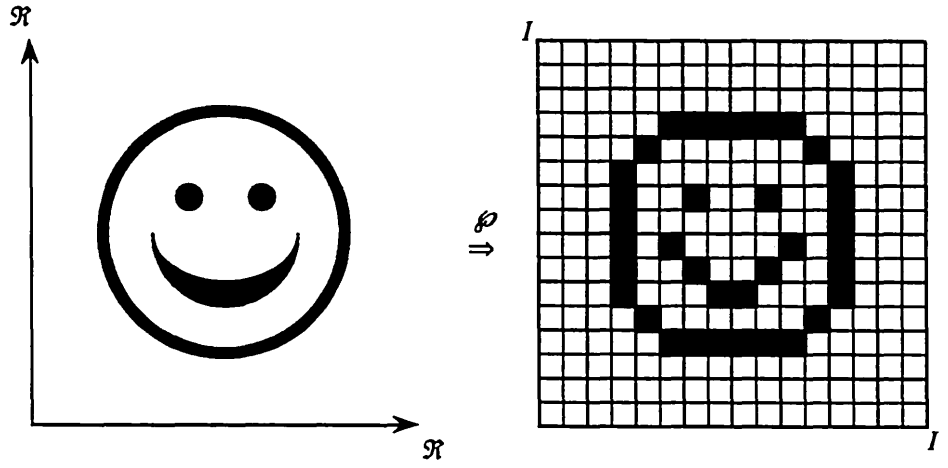


fig. 3.8

"adjacent" pixels (i.e. those pixels reached by applying all transforms to the current location). If a message is received by a pixel which is already `lit`, its state does not change. On the other hand, if the pixel is `off`, then it turns itself `on` so that it will send messages to its adjacent nodes in the next iteration. The function which computes the next state of a pixel given the list of arriving messages (`xs`) and the current state (`st`) is `nextstate`:

```
nextstate = λxs.λst.
   case xs in
      when cons h t :
         case st in
            when off : on
            otherwise : st
      when nil : st
```

All pixels send messages to their adjacent pixels. Those pixels that were `on` then change state to `lit`. Once a pixel is `lit`, it stays `lit`. There is redundancy in this program in that `lit` pixels do not need to send messages to their successors because they have already been switched `on`. Therefore, only the messages sent by pixels which have just been turned `on` are necessary. All other messages are redundant (and only serve to burden the interconnection network). Ideally, PEs corresponding to `off` or `lit` pixels should be excluded from subsequent communication. This cannot be done using the communication operations presented so far since they are *unconditional* (i.e. all elements must transmit). This issue is addressed in more detail in §3.2. The main part of the communication and computation is performed by the function `sendon`:

```
changestate = λst.λnt. (= st on) → lit; nt

sendon = λar.λdrs.
   map2 changestate ar
      (fold (λdr.λbr. map2 nextstate (send dr br) br) ar drs)
```

Transformations Applied to a Pixel

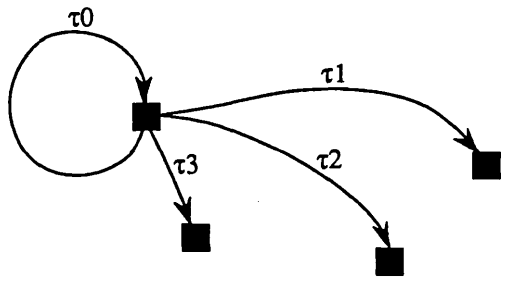


fig. 3.9

The array `ar` contains the current state of the pixels in the display. The argument `drs` is a list of arrays of destination indices giving the locations of the adjacent pixels for each pixel. The `fold` function was used in the definition of `scatter` in the previous chapter. For each array of destinations, a message is sent to an adjacent pixel. This is repeated each iteration until all the pixels in the image have been traversed. This condition occurs when the array of pixel states no longer changes (i.e. pixels are either `lit` or `off` and no pixels are `on`). The function `iterate` applies `sendon` until the array of pixel states does not change (i.e. all nodes in the graph have been visited):

```
iterate = λar.λdrs.
   let br = sendon ar drs in
   let eq = reduce (∧) (map2 (=) ar br) in
      (= eq true) → ar; iterate br drs
```

The list of destination arrays (`drs`) is computed by taking the list of transforms (consisting of *a*, *b*, *c*, *d*, *e*, and *f* values for each transform), and the index of each array

element as the projection of a point (x, y) into the integer coordinate space $(I \times I)$. The destination (i.e. an adjacent node) can then be computed as the application of the affine transform to the index of the array element.

A sample IFS image consisting of only 4 transforms is given in fig. 3.11. This

Image as Connected Pixels in a Graph

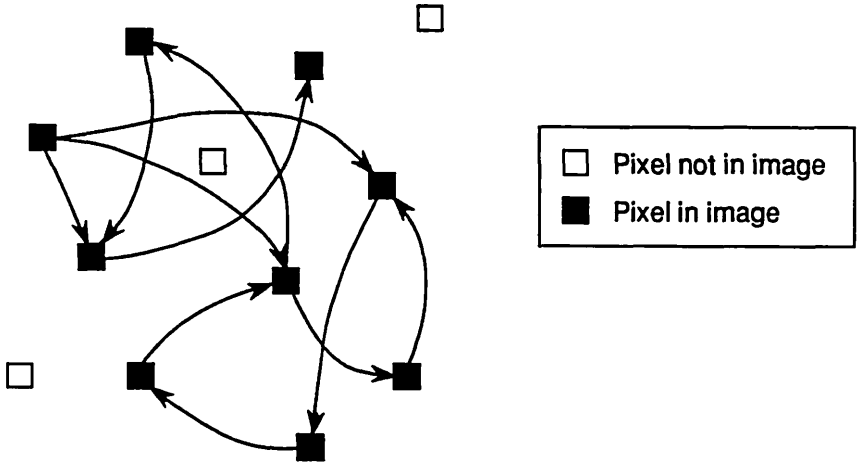


fig. 3.10

Sample Fractal Image (Fern Leaf on 225 ×225 grid)

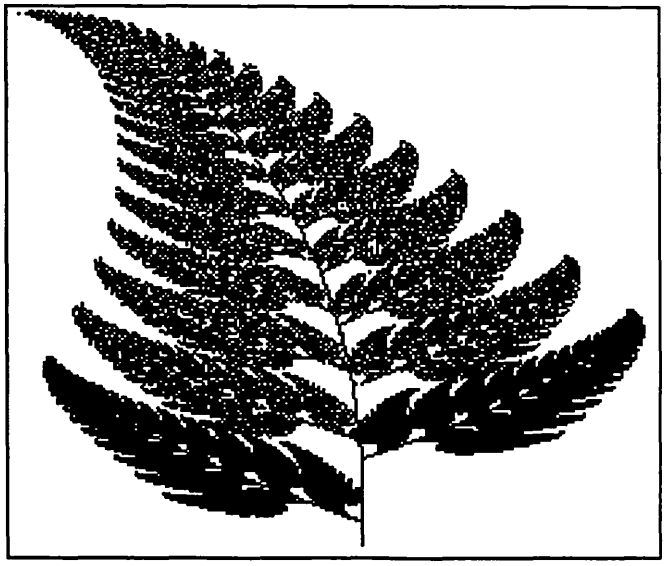


fig. 3.11

image was projected on a display of 225×225 pixels. If no contention exists for messages sent to the same destination, then the degree of parallelism in the IFS program can be shown as the number of on locations per iteration. For the case of the fern in

fig. 3.11, 33 iterations were necessary to traverse all the pixels in the image. The average parallelism in terms of the active elements per iteration is shown in fig. 3.12.

This program is particularly interesting because it shows that graph algorithms can readily be implemented in data-parallel fashion. A wide variety of applications have been shown to be isomorphic to problems in graph theory and the discover the decompression of IFS images is equivalent to the problem of traversing a gra rting from the fixpoints of the transformations is new. This program, as the prev two before it, relies heavily on the availability of data-parallelism in the simultaneous movement of data-elements.

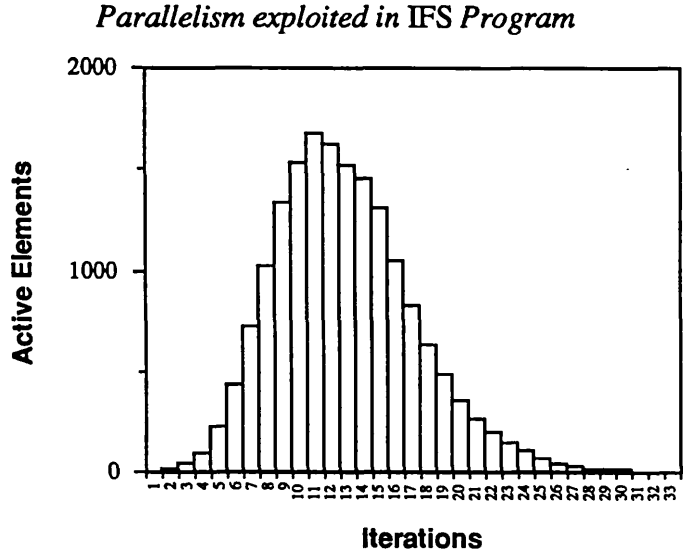


fig. 3.12

3.2. Observations

Some of the programs presented in this chapter were exceptionally concise and clearly exploited the expressive power of higher-order operations provided by functional languages. Others, such as the Gaussian elimination example are significantly more cumbersome than the equivalent algorithm expressed in conventional imperative languages. To make this comparison fair it must be noted that the conventional imperative form of this algorithm contains little inherent parallelism. The bulk of the complexity in this particular example is due to the strict notions of intra-element locality enforced by a data-parallel model of computation: all needed values must explicitly be communicated. Many numerical algorithms have an underlying regular access pattern to data elements so it may be possible to capture a subset of these common forms in a library to facilitate the expression of algorithms involving lots of communication. In Chapter 7 this particular topic is addressed in detail in the context of Parallel Data Transforms which show how

transformation techniques can be used to facilitate the development of fast and efficient communication patterns.

In this section some of the implications of the programs presented in this chapter are discussed. One source of potential awkwardness is the lack of conditional communication. The communication primitives: `rotate`, `send`, and `fetch` are unconditional: all elements are communicated to their destination. In the case of the digital circuit simulation example (§3.1.4) the inputs to the original circuit were left unchanged. Therefore, it would be convenient to have a conditional `send` operation which excluded these elements from communication and it would not be necessary to introduce the "fix" provided by routing the inputs back to themselves and applying the `fst` operation. Also, the Gaussian elimination example (§3.1.5) and the IFS algorithm (§3.1.6) would be simplified by the introduction of a conditional communication primitive. In the case of IFS, those pixels which have been reached by communication would thereafter be excluded from further communication. A conditional `send` to an inactive location would have no effect. A further advantage of conditional communication primitives is that they reduce network traffic since fewer messages are sent which reduces network contention in global communication.

The introduction of conditional communication introduces additional problems however. Bougé has developed two operational semantics for a simple data-parallel language called $\mathcal{L}$ [Boug91]. These two semantic models correspond to what he calls the *macroscopic* and *microscopic* views of data-parallel machines. The first corresponds to the view of such machines as "processors-of-arrays," the second as an "array-of-processors." The first semantic model is monolithic, the second is element-wise. Bougé discovered that the introduction of conditional communication operations renders these two semantics of $\mathcal{L}$ unequal. This has serious implications for compilation of the language augmented by the data-parallel primitives presented in the previous chapter. Note how the `imap` primitive takes a functional argument which uses element-wise arithmetic/logical and control constructs. The compilation system (to be introduced in Chapter 5) translates these element-wise constructs to monolithic abstract machine instructions, in effect converting from a microscopic to a macroscopic language semantics. If there is no guaranteed equivalence between these two semantics, then the translation performed by the compilation is unsafe. Therefore, conditional communication operations cannot be allowed in our system because the compilation system could not preserve the semantics of the source-level program.

Using the data-parallel primitives provided is not significantly more difficult than using any other types of pre-defined functions. A minimal set of simple primitives actually *aid* algorithm development because the programmer has to decompose a problem to a form that uses only these pre-defined operations. The process of parallel algorithm-

development is facilitated because the range of options is narrowed. This is a positive aspect in that as experience with using the operations is gained it becomes easier to recognize familiar sub-structures in algorithms which can readily be captured by a particular data-parallel operation.

Chapter 4

4. An Abstract Data-Parallel Machine Architecture

Having specified a set of operations to exploit data-parallelism at the language level, it is necessary to specify the target architecture for their execution. An abstract data-parallel machine based on the SIMD model is developed, categorized by its native instruction set and its operational semantics defined via transitions on the machine state.

Initially, a variant of the Spineless-Tagless G-Machine (STGM), a scalar abstract architecture for the evaluation of lazy functional languages is adopted. The abstract data-parallel machine model (PAM) can then be described in terms of the modifications required of the STGM. The development of PAM proceeds in two stages: specification of a *virtual* abstract machine by the addition of a planar memory mode to the STGM, followed by a refinement to a *concrete* abstract architecture which implements the virtual model. The concrete abstract machine addresses the limits of finite processing and memory resources that exist in any real machine.

PAM employs a dual scalar/planar memory model and provides a suitable abstraction of currently available SIMD architectures. The implicit manipulation of activity masks to enable conditional execution and recursion, the use of closures to support a lazy semantics, the use of algebraic data-types as array elements, the maintenance of index planes, and the tiling of memory planes to provide support for dynamically allocated arrays of variable size are discussed in turn.

The role of this abstract machine architecture is twofold: to serve as a target form for compilation and as an operational model for the possible implementation of PAM on MIMD machines.

4.1. Purpose of Abstract Machines

Modern compilers typically use an intermediate form before compiling a source language into the language of the target architecture. There are several advantages to this approach: optimizations can be performed more easily on the intermediate form, code generation is made easier, and generality across specific platforms is retained as the intermediate form can be made independent of the target architecture.

4.1.1. Abstract Architectures for Functional Languages

Intermediate forms for functional languages can be entire languages in their own right or merely abstractions of a desired target architecture. Existing intermediate languages used in the compilation of functional languages for parallel architectures contain features which allow for the exploitation of the process-parallel model of computation and address the specific aspects relevant to this model such as: process granularity, parallel task evaluation, and data sharing. Examples include FLIC [Peyton88], developed for the GRIP parallel Machine, P-TAC [Ariola89], developed for the MIT Monsoon Dataflow project, and DACTL [Glauer87] (subsequently developed into CLEAN), originally designed as an intermediate language for the implementation of both logic and functional languages on the Alvey Flagship project. For uniprocessor implementations, these languages are too abstract and compilation efforts on sequential machines are usually directed towards an abstraction of a conventional von-Neumann processor.

These abstract architectures for functional languages can be divided into two types: environment and stack-based. In the first scheme, the values associated with variables are stored in an *environment.* An early example of this type of architecture is Landin's SECD machine [Landin64]. A *stack-based* architecture places all needed variables onto the stack. This requires that programs be *lambda-lifted* (which converts lambda-expressions to supercombinators) to remove all free-variables in expressions. Examples of this include the G-Machine [Johnss84] and the TIM [Fairba87]. Lazy evaluation schemes rely on building *closures*, or suspended evaluations, which contain pointers to code and a local environment for storing pointers to applied arguments. These closures can then be evaluated whenever their values are demanded.

Interestingly, recent architectures are hybrids of these two extremes. The Functional Programming Machine, or FPM [Bailey85] (which forms the abstract-machine for the Hope$^+$ language [Perry88]) and its derivatives is a stack-based variant of the SECD machine. The most recent variation on the G-Machine, the STGM [Peyton89], provides an environment in which free variables can be placed. The availability of an environment in a stack-based system means that it is no longer necessary to perform lambda-lifting as free-

variables of lambda-expressions can be stored in the environment. The elimination of lambda-lifting has the benefit of further simplifying the compilation process.

4.1.2. Development of an Abstract Data-Parallel Machine

A SIMD realization of data-parallel computation consists of a single stream of (parallel) instructions, manipulated by a single independent control unit. Such an architecture therefore has more in common with conventional scalar von-Neumann processors than MIMD systems where a number of processors indulge in independent (and potentially distinct) activities. The use of an architecture abstraction as an intermediate form is therefore more suited to our purposes than an explicitly parallel intermediate-language. A simplified STGM is used as the starting point for the development of an abstract data-

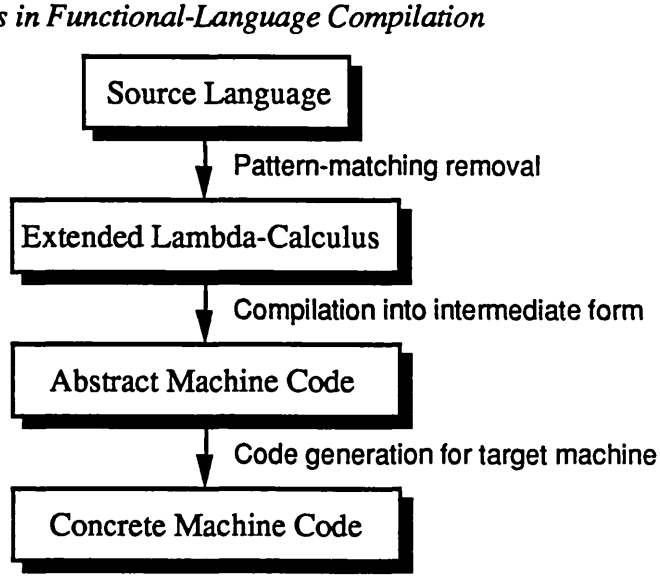


fig. 4.1

parallel machine.

4.2. A Scalar Abstract Machine: Simplified STGM

Compilation of a functional language for a specific architecture proceeds through a number of stages, starting from the source language, removing syntactic features such as pattern-matching, compiling to the abstract machine code, followed by a code-generation phase to produce actual machine code for the target architecture. These stages are depicted in fig. 4.1. The language presented in Chapter 2 is the extended lambda-calculus to which most functional languages are transformed as an initial step in the compilation effort.

In specifying the abstract architecture, some of the more low-level aspects of the machine such as the optimization of storage of basic values, maintenance of update-

markers to update references to shared expressions, etc will be ignored. Work on these aspects developed for conventional von-Neumann architectures remains applicable to the extensions developed here but is not directly relevant. In this chapter only those aspects of the abstract architecture which are necessary for the exploitation of data-parallelism are discussed.

In order to focus on only the core concepts of the STGM, a simplification of the machine model is used. This reduced abstract machine, the simplified STGM (SSTGM), nevertheless retains the capabilities of the STGM, without the complexity due to implementation optimizations.

The SSTGM differs from the STGM in the following areas:

- No lambda-lifting needs to be performed as free variables can be stored in the closure environment.
- The basic instruction set has been simplified.
- The machine state has been reduced (no dedicated register is set aside for discriminators or returned basic values).
- Support for `letrec` (recursive `let`) has been omitted.

The SSTGM places pointers to all arguments to lambda-expressions on a stack. Arguments to functions are referenced as offsets from the stack pointer. Objects in the heap may consist of algebraic data-types, basic values, or suspended evaluations (introduced by normal-order reduction). An attractive property of the STGM is that there is a uniform representation for all of these objects, namely as *closures*. Closures consist of a pointer to code, followed by pointers to variables. In tagged-implementations, each object in the heap is explicitly tagged with a label to denote the type of the object. Operations on heap objects must always check the tag so that an appropriate action may be performed. The STGM replaces these tags with a pointer to code. Instead of determining the type of an object by checking the tag and then taking some appropriate action, the STGM merely branches to the code pointer in every closure which carries out the appropriate action.

A dedicated register points to the closure currently being evaluated. Free variables are accessed as offsets from this pointer. Evaluation of a closure is performed by entering its code address. This simple evaluation mechanism is one of the principal attractions of the STGM and will be shown to be especially useful for the abstract data-parallel machine presented in the following section.

4.2.1. SSTGM Instructions

An abstract machine can be categorized by a tuple denoting its state and a set of operations defined in terms of transitions on this state.

The state of the SSTGM is denoted by the tuple $\langle C, H, S, E, N \rangle$ consisting of:

- The stream of code instructions, denoted by C.
- A heap for allocating closures, H.
- A stack for arguments to functions, S.
- An environment, E, which consists of the named locations of arguments on the stack, static code and data space, as well as the free-variables accessible as offsets from the location pointed to by N.
- A dedicated node register which points to the current closure under evalutation, N.

The code stream, heap, and stack can be viewed as linear structures, decomposed and constructed by the use of the infix `cons` (::) constructor. A closure is represented by a tuple: $(code, var_1, \ldots, var_n)$ where *code* denotes the sequence of instructions that will be executed when the closure is entered and $var_1, \ldots, var_n$ denote the variables accessible by *code*.

Algebraic data-types are also represented as closures. Given the following data-type, `tree` (specified using the curried constructor form):

```
tree α = empty | leaf α | node (tree α) (tree α)
```

The constructor identifiers are compiled into *discriminators* (ordinal values) and objects of type `tree` are closures: $(code, disc, var_1, var_2)$, where *code* returns the discriminator, *disc*, and var_1, var_2 are pointers to the constructor arguments of `leaf` and `node` terms of `tree`.

It is wasteful to allocate an entire closure to represent objects of the basic types: *Int*, *Char*, *Bool*, and discriminators (which are just ordinal values). Therefore, variables of basic values are usually *unboxed* (the term used by Peyton-Jones) and stored directly in the memory location denoted by the variable. Variables representing other objects are pointers to closures. Unboxed variables contain the object itself. Using an unboxed representation for basic values avoids an indirection, saves heap space, and allows these variables to be allocated to registers for fast access.

The SSTGM instructions are defined informally in fig. 4.2. The instructions are also defined formally in terms of transitions on the machine state in fig. 4.3

SSTGM Instructions

x	Variable, function, or constructor
u	Unboxed basic value
c	Constructor discriminator
p	Primitive operation
f	Function
t	Closure

u := BASIC_OP (p) [u_1, ..., u_n]	Apply the basic operation p to the values in the unboxed variables u_1, ..., u_n, storing the result in u.
u := EVAL {*code*}	Execute the instructions in *code*, which will return an enumerated basic value, to be stored in u.
f := COMBINATOR {*code*}	Define a new combinator in static code space.
t := CLOSURE [x_1, ..., x_n] {*code*}	Construct a closure (*code*, x_1, ..., x_n) in the heap and point t to it.
RETURN_BASIC u	Return the enumerated basic value stored in unboxed variable u to the location residing on the top of the stack.
RETURN_CONSTR c [x_1, ..., x_n]	Return the discriminator c to the location residing on the top of the stack, construct a closure (RETURN_BASIC (c), c, x_1, ..., x_n) in the heap and point N to it.
ENTER x [x_1, ..., x_n]	Place x_1, ..., x_n on to the stack and branch to the address pointed to by the *code* field of the closure pointed to by x.
POP x	Associate x with the top of the stack and decrement the stack pointer.

CASE {
alt_1
...
alt_n
}
	Transfer execution to one of the continuations alt_1, ..., alt_n, if the corresponding constructor matches the discriminator returned by the evaluation of the predicate expression.

WHEN (c, u) [x_1, ..., x_n] {*code*}	Execute *code* if the discriminator, u, matches the constructor c. Constructor variables x_1, ..., x_n can be referenced as offsets from N.
OTHERWISE {*code*}	Execute *code* if no previous alternative is matched.

fig. 4.2

The CASE instruction uses the discriminator returned by evaluation of the predicate expression to transfer control to the matching continuation. Variables in the environment (E) may consist of explicit arguments to functions, temporary variables, and free variables accessible via N. All free variables and constructor variables are accessed as offsets from N, whereas function arguments and temporary variables reside on the stack.

Top-level functions are compiled to *combinators* (sequences of instructions corresponding to functions) stored in the code-segment area of the heap. Calls to functions are performed by jumping directly to the location of the combinator in static code space. There are in effect two variants of the ENTER instruction: one for closures and one for functions. Wherever ENTER is used with a user-defined function, the code-address of the function is known at compile-time and this is implemented as a direct jump. If a closure is entered the code address resides in a variable and hence an indirect jump needs to be performed.

State Transition Table for SSTGM Instructions

$\langle$(*u* := BASIC_OP (*p*) [u_1, ..., u_n]::*code*)::*C*, *H*, *S*, *E*, *N*$\rangle$
$\Rightarrow \langle$*code*::*C*, *H*, *S*, *E*[(*p* *E*[u_1], ..., *E*[u_n])/*u*], *N*$\rangle$

$\langle$(*u* := EVAL {$code_1$}::*code*)::*C*, *H*, *S*, *E*, *N*$\rangle$
$\Rightarrow \langle$$code_1$::*code*::*C*, *H*, *u*::*E*::*S*, *E*, *N*$\rangle$

$\langle$(t_1 := CLOSURE [x_1, ... x_n] {$code_1$}::*code*)::*C*, *H*, *S*, *E*, *N*$\rangle$
$\Rightarrow \langle$*code*::*C*, t_2::*H*, *S*, *E*[t_2/t_1], *N*$\rangle$ $\quad t_2 = (code_1, E[x_1], ..., E[x_n])$

$\langle$(RETURN_BASIC u_1::[])::*C*, *H*, u_2::*e*::*S*, *E*, *N*$\rangle$
$\Rightarrow \langle$*C*, *H*, *S*, *e*[*E*[u_1]/u_2], *N*$\rangle$

$\langle$(RETURN_CONSTR *c* [x_1, ..., x_n]::[])::*C*, *H*, *u*::*e*::*S*, *E*, *N*$\rangle$
$\Rightarrow \langle$*C*, *t*::*H*, *S*, *e*[*c*/*u*, *E*[x_1]/cx_1, ..., *E*[x_n]/cx_n], *t*$\rangle$
$\quad t$ = (RETURN_BASIC *c*::[], *c*, *E*[x_1], ..., *E*[x_n])

$\langle$(ENTER *x* [x_1, ..., x_n]::[])::*C*, *H*, *S*, *E*, *N*$\rangle$
$\Rightarrow \langle$*code*::*C*, *H*, x_1::...::x_n::*S*, $\perp$[*E*[y_1]/z_1, ..., *E*[y_m]/z_m], *t*$\rangle$ $t = E[x] = (code, y_1, ..., y_m)$

$\langle$(POP x_1::*code*)::*C*, *H*, x_2::*S*, *E*, *N*$\rangle$
$\Rightarrow \langle$*code*::*C*, *H*, *S*, *E*[x_1/x_2], *N*$\rangle$

$\langle$(CASE {alt_1, ..., alt_n}::[])::*C*, *H*, *S*, *E*, *N*$\rangle$
$\Rightarrow \langle$[alt_1, ..., alt_n]::*C*, *H*, *S*, *E*, *N*$\rangle$

$\langle$(WHEN (*c*, *u*) [cx_1, ..., cx_n] {*code*}::*alts*)::*C*, *H*, *S*, *E*, *N*$\rangle$
$\Rightarrow \langle$*code*::*C*, *H*, *S*, *E*, *N*$\rangle$ $\quad$ if $(E[u] = c)$
$\Rightarrow \langle$*alts*::*C*, *H*, *S*, *E*, *N*$\rangle$ $\quad$ otherwise

$\langle$(OTHERWISE {*code*}::[])::*C*, *H*, *S*, *E*, *N*$\rangle$
$\Rightarrow \langle$*code*::*C*, *H*, *S*, *E*, *N*$\rangle$

fig. 4.3

The state transitions define an operational semantics for the SSTGM instructions in terms of their effect on the machine state. In keeping with the high-level scheme of the STGM, the state-transitions defer a number of implementation aspects to the concrete-machine level. Some of these aspects are detailed in the following section.

4.2.2. Note on State Transitions

Use is made of the environment to represent variables within the current "scope." An analogous notion is the use of a *stack-frame* in imperative language implementations. In a concrete machine implementation, temporary variables are stored on the stack or in registers. This requires that the ENTER instruction place its arguments on the stack and then "squeeze" out the no-longer required temporary variables on the stack. In the state

transitions in fig. 4.3, the locations of temporary variables are not specified, except that variables are made accessible to subsequent instructions (by being placed in the environment). The benefit of this approach is that the semantics remains uncluttered with unnecessary implementation-specifc aspects (for example, it is not necessary to show squeezing in the SSTGM).

The instructions in an EVAL block retain access to all temporary variables created in previous instructions and may create additional variables. Eventually a EVAL block returns to its caller by executing a RETURN_BASIC or RETURN_CONSTR instruction at which point it is possible to discard all temporary variables created in the the EVAL block. In the state transitions, this is shown by pushing the existing environment onto the stack ($E::S$) when executing an EVAL, and restoring this saved environment when executing a RETURN_BASIC or RETURN_CONSTR.

In addition to arguments popped off the stack and temporary variables, free and constructor variables are accessible in the closure pointed to by N. The action of setting N to point to a closure makes such variables accessible in the concrete machine. In the state transition for the RETURN_CONSTR instruction, this is shown explicitly by adding the constructor variables to the saved environment, e: $e[c/u, E[x_1]/cx_1, \ldots, E[x_n]/cx_n]$. In the case of the ENTER instruction where free variables may come into scope, these are also explicitly shown added to a new, empty, environment (denoted by $\perp[]$) $\perp[E[y_1]/z_1, \ldots, E[y_m]/z_m]$. The labels $z_1, \ldots z_n$ are used to represent the free variables which are being brought into scope.[1]

Evaluating the predicate expression of a `case`-statement causes the N register to be overwritten which causes free variables in the continuations to become inaccessible. This is prevented by placing these variables explicitly on the stack prior to transfering control. It is assumed that the subsequent stage in the compilation which maps the STGM onto a concrete target architecture takes care of this.

It is important to keep clear the distinction between the use of an environment as an artifice for *abstraction* purposes versus the use of an environment as an implementation device. In the SSTGM it is convenient to employ an environment in order to present a concise semantics of the machine instructions. At the concrete machine level, the environment is distributed between arguments residing on the stack, temporary variables created on the stack or in registers, and free or constructor variables in closures accessible via N.

---

[1] In any concrete implementation of course, no labels are actually used. Instead, free variable identifiers are commonly converted into de Bruijn numbers.

4.2.3. Compilation Scheme to Scalar Code

The following compilation rules (see fig. 4.4) illustrate the translation from our initial language to scalar abstract machine code. The compilation scheme uses a compile-time stack denoted by σ. This stack is used to accumulate arguments to applications. If lambda-expressions are applied to arguments present at compile-time, then the argument is brought into scope by being "popped-off" the compile-time stack. Otherwise, an explicit run-time `POP` instruction is generated. A program is a series of definitions $f_1 = e_1 \ldots f_n = e_n$ followed by a top-level expression, e.

Compilation to SSTGM Code

k	Constant
x	Variable, function, or constructor
u	Unboxed basic value
c	Constructor discriminator
p	Primitive operation
f	Function
t	Closure

$\mathcal{T}[\![k_1 = e_1 \ldots k_n = e_n; e]\!] =$
k_1 := `COMBINATOR` $\{\mathcal{E}[\![e_1]\!]\ []\}$
...
k_n := `COMBINATOR` $\{\mathcal{E}[\![e_n]\!]\ []\}$
$\mathcal{E}[\![e]\!]\ []$

$\mathcal{E}[\![k]\!]\ \sigma =$ $u := k$; `RETURN_BASIC` u
$\mathcal{E}[\![c]\!]\ \sigma =$ `RETURN_CONSTR` $c\ \sigma$
$\mathcal{E}[\![p\ e_1\ e_2]\!]\ [] =$ $\mathcal{B}[\![p\ e_1\ e_2]\!]\ u$; `RETURN_BASIC` u
$\mathcal{E}[\![x]\!]\ \sigma =$ `ENTER` $x\ \sigma$
$\mathcal{E}[\![\lambda x.e\ ]\!]\ t::\sigma =$ $x := t$; $\mathcal{E}[\![e]\!]\ \sigma$
$\mathcal{E}[\![\lambda x.e\ ]\!]\ [] =$ `POP` x; $\mathcal{E}[\![e]\!]\ []$
$\mathcal{E}[\![e_1\ e_2]\!]\ \sigma =$ $t := C[\![e_2]\!]$; $\mathcal{E}[\![e_1]\!]\ t::\sigma$
$\mathcal{E}[\![\texttt{let}\ x_1 = e_1\ \texttt{in}\ e_2]\!]\ \sigma =$ $x_1 := C[\![e_1]\!]$; $\mathcal{E}[\![e_2]\!]\ x_1::\sigma$
$\mathcal{E}[\![e]\!]\ \sigma$
$\mathcal{E}[\![\texttt{case}\ e\ \texttt{in}\ a_1 \ldots a_n]\!]\ \sigma =$ u := `EVAL` $\{\mathcal{E}[\![e]\!]\ []\}$
`CASE` {
$\mathcal{A}[\![a_1]\!]\ \sigma\ u$
...
$\mathcal{A}[\![a_n]\!]\ \sigma\ u$
}

$C[\![k]\!] =$ k
$C[\![x]\!] =$ x
$C[\![e]\!] =$ `CLOSURE` $\mathcal{FV}[\![e]\!]\ \{\mathcal{E}[\![e]\!]\ []\}$

$\mathcal{A}[\![\texttt{when}\ c\ x_1 \ldots x_n : e]\!]\ \sigma\ u =$ `WHEN` $(c, u)\ [x_1 \ldots x_n]\ \{\mathcal{E}[\![e]\!]\ \sigma\}$
$\mathcal{A}[\![\texttt{otherwise} : e]\!]\ \sigma\ u =$ `OTHERWISE` $\{\mathcal{E}[\![e]\!]\ \sigma\}$

$$
\begin{array}{lcl}
\mathcal{B}[\![k]\!]\ u & = & u := k \\
\mathcal{B}[\![p\ e_1\ e_2]\!]\ u & = & \mathcal{B}[\![e_1]\!]\ u_1 \\
 & & \mathcal{B}[\![e_2]\!]\ u_2 \\
 & & u := \texttt{BASIC\_OP}\ (p)\ [u_1, u_2] \\
\mathcal{B}[\![e]\!]\ u & = & u := \texttt{EVAL}\ \{\mathcal{E}[\![e]\!]\ []\}
\end{array}
$$

fig. 4.4

The function $\mathcal{FV}[\![e]\!]$ returns the set of free variables in the expression e. The $\mathcal{T}$ scheme is used for the compilation of programs, $\mathcal{E}$ for expressions, $\mathcal{C}$ for closures, $\mathcal{A}$ for `case`-continuations, and $\mathcal{B}$ for strict built-in operations.

Compilation schemes conventionally replace all identifiers with an internal representation which avoids the possibility of name clashes. This has not been done here in order to preserve the simplicity of the compilation scheme. In addition, the readability of compiled code examples is enhanced since original identifiers are preserved.

4.2.4. Scalar Code Generation Examples

The compilation rules in the previous section can best be understood through a series of simple examples. Examples 4.1 and 4.2 are taken from the original STGM paper [Peyton89]:

Example 4.1 Consider the following definition of the compose function:

```
compose = λf.λg.λx. f (g x)
```

This compiles to the following scalar code:

```
comp := COMBINATOR {
   POP f
   POP g
   POP x
   t1 := CLOSURE [g, x] { ENTER g [x] }
   ENTER f [t1]
}
```

The arguments `f`, `g`, and `x` will be on stack when `compose` is invoked. They are subsequently brought into scope by being popped off the stack. A closure is constructed for the application of `g` to `x`. The free variables of this expression (i.e. `g` and `x`) are put in the environment for the closure.
□

The following example illustrates how arithmetic operations are compiled:

Example 4.2 Consider the arithmetic function:

```
arith = λx.λy. (- (* y y) x)
```

This produces the following scalar code:

```
arith := COMBINATOR {
   POP x
   POP y
   u1 := EVAL { ENTER y [] }
   u2 := EVAL { ENTER y [] }
   u3 := BASIC_OP (*) [u1, u2]
   u4 := EVAL { ENTER x [] }
   u5 := BASIC_OP (-) [u3, u4]
   RETURN_BASIC u5
}
```

The fact that subtraction is a known strict operation (i.e. requires the values of both of its arguments) means its arguments can be evaluated directly without building closures for them by using the $\mathcal{B}$ compilation rule.
□

The ubiquitous factorial function is presented below:

Example 4.3 This is the tail-recursive rendition of factorial:

```
fact = λn.λa. (= n 0) → a; fact (- n 1) (* n a)
```

Which compiles to:

```
fact := COMBINATOR {
   POP n
   POP a
   u1 := EVAL {
      u2 := EVAL { ENTER n [] }
      u3 := 0
      u4 := BASIC_OP (=) [u2, u3]
      RETURN_BASIC u4
   }
   CASE {
      WHEN (true, u1) [] { ENTER a [] }
      WHEN (false, u1) [] {
         t5 := CLOSURE [a, n] {
            u6 := EVAL { ENTER n [] }
            u7 := EVAL { ENTER a [] }
            u8 := BASIC_OP (*) [u6, u7]
            RETURN_BASIC u8
         }
         t9 := CLOSURE [n] {
            u10 := EVAL { ENTER n [] };
            u11 := 1;
            u12 := BASIC_OP (-) [u10, u11];
            RETURN_BASIC u12
         }
         ENTER fact [t9, t5]
      }
   }
}
```

This version of the factorial function is inefficient in that it builds unnecessary closures for the expressions `(- n 1)` and `(* n a)`. The use of effective strictness analysis techniques can discover that `fact` is strict in both of its arguments (i.e. will always evaluate `n & a`) so that it is unnecessary to build closures to delay their evaluation. This optimization is addressed in Chapter 7.
□

Having gained an understanding a conventional scalar abstract machine instructions and the compilation scheme to generate these instructions from our functional language the modifications necessary to support a data-parallel mode of execution can now be described. The extensions consist primarily of an additional memory mode to permit instructions to execute on multiple words of data concurrently.

4.3. The Virtual Planar Abstract Machine: PAM

A data-parallel architecture has two memory modes: scalar and planar. In the scalar mode, operations apply to scalar values and is equivalent to a conventional von-Neumann processor. The planar mode is essentially a *super-scalar*, or *multi-word*, extension which allows a single operation to apply to multiple words (or *plane* of words) concurrently. Adding a planar memory mode to the SSTGM yields the *virtual planar abstract machine*, or PAM. In PAM, virtual planes of any size can be declared. This model will subsequently be refined to a *concrete planar abstract machine*, or CPAM, which is limited to fixed-size planes.

In SIMD architectures the planar mode is implemented by replicating the execution unit of a conventional processor to yield a mesh of processing elements (PEs). Planar instructions are decoded centrally by a scalar central processing unit (CPU) and broadcast to the distributed PEs which carry out the instruction on values in their local stores. A plane corresponds to an individual word at the same address in the store of each PE. PAM can be viewed as an abstraction of SIMD architectures (see fig. 4.5) where restrictions on the number of available PEs or resource limitations (e.g. memory) are ignored. This diagram shows a 2-dimensional abstract machine model but the definition of a plane scales to data-parallel architectures of higher-dimensions.

PAM is a virtual architecture in that it assumes that the number of PEs can be varied dynamically. The planar instructions in PAM operate on *virtual plane* operands. The size of virtual planes can be declared at run-time and need not coincide with the number of PEs in the concrete architecture. Large virtual planes are emulated by the concrete machine by collections of planes of fixed size which are "tiled" to represent the virtual plane (see §4.4.1). The relationship between planes in the abstract machine and arrays in the source language is explained in §4.3.3.3.

4.3.1. Scalar/Planar Dichotomy

PAM's native planar instructions exhibit data-parallelism but the programmer cannot be expected to write programs directly in PAM-code. Programmers have access to data-parallel computation through the `imap` primitive, but the argument to `imap` consists of ordinary, scalar, functions. In order to exploit data-parallelism in these functions, the compilation scheme must generate data-parallel variants of all user-defined functions which

may be used as arguments to `imap`. This requires that the compilation scheme be capable of translating all arbitrary control-structures and data-structures available in the extended lambda-calculus to a form appropriate for data-parallel execution. This provides significant flexibility because existing functions tested on scalar values can be mapped directly on to arrays to exploit data-parallelism. The exploitation of data-parallelism therefore does not require the programmer to learn and compensate for the restrictions of the target architecture.

4.3.1.1. Dual Entry Points

Any non-array function may be used as an argument to `imap`. In general, determining which of the functions in a program will be used as arguments to `imap` is undecidable due

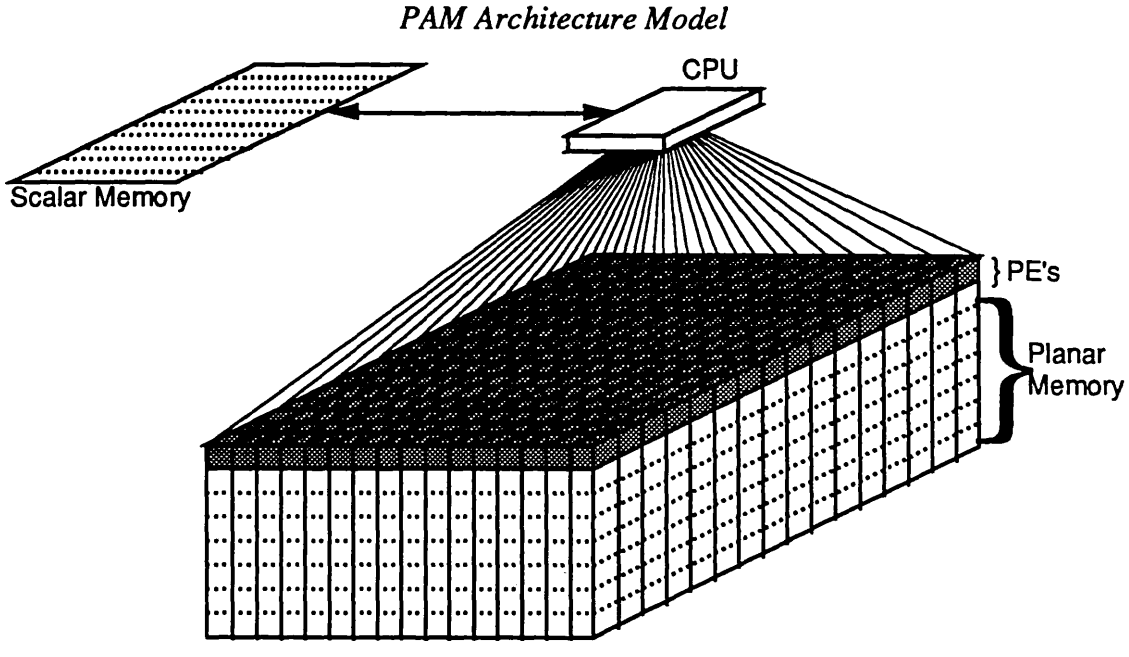


fig. 4.5

to the higher-order capability of our language. Therefore, two versions of every function are created: one operating on scalar operands, the other on planar ones. Each combinator definition therefore needs two entry points, corresponding to the two different code streams for the function. Two different kinds of ENTER instruction are provided to correspond to provide access to the appropriate entry point.

Providing multiple entry points for functions is not new. Such a scheme is often used in conjunction with strictness analysis to create alternative renditions of functions which do not try to evaluate arguments that have already been evaluated, as in the *Evaluation Transformer Model* [Burn87]. In this scheme 4 different entry points for functions taking list-valued arguments are created. These entry points cause list-arguments

to be evaluated differently (i.e. not at all, head-strict, tail-strict, and fully-strict). The use of separate entry points in PAM is quite distinct from previous applications since the two entry points correspond to completely different types of instruction streams (scalar v.s. planar).

4.3.2. Memory Areas

Memory in PAM is divided into two areas: scalar and planar (see fig. 4.6). Scalar memory is divided into word-sized units and is organized into several main areas: static code space, constant space, the heap, and the stack. All functions are compiled into streams of code stored in code space. Both scalar and planar instructions are stored here. Scalar constants are stored in the constant space. The heap is used for the allocation of closures and algebraic data-types. There are two types of suspended evaluations, corresponding to the

PAM Memory Organization

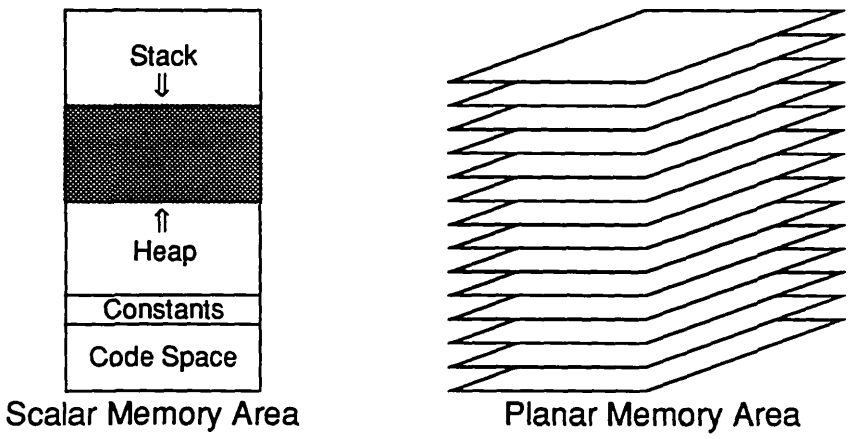


fig. 4.6

delayed evaluation of expressions in the scalar and planar code streams. Both types of closures reside in the scalar heap. Unboxed scalar values reside on the scalar stack. The planar memory area is only used for the storage of basic (unboxed) planar values.

4.3.3. Objects

The storage and representation of the various types of objects stored in the memory areas of PAM are discussed next. These objects are basic values, closures, and algebraic data-types.

4.3.3.1. Basic Values

The basic types include: *Int*, *Bool*, *Char*, *Real*, and the discriminators corresponding to the compiled constructor identifiers. These objects are word-size (or less) and are stored unboxed (i.e. not as a closure) as in the SSTGM.

The only objects stored in the planar memory area are basic planar values. Basic planar values also have a *dimension* associated with them, in other words, their size (corresponding to the number of elements in an array). For example, a virtual plane can be 100 elements large (if it represents a one-dimensional array), 200 × 100 (in which case it represents a two-dimensional array), or any other size (subject to memory limitations).

Basic operations are applied to scalar operands by the `SCALAR_OP` instruction, to planar operands via `PLANAR_OP`. The instruction:

```
w3 := PLANAR_OP (+) [w1, w2]
```

Adds the elements in planes `w1` and `w2` together (assigning the result to `w3`) and is a constant-time operation. Planar operations can only be applied if all the operands have the same dimensions.

Planar arguments on the stack are actually pointers to plane descriptors with the following fields:

(b, vp, rc) :
b = Array bounds
vp = Virtual plane representation
rc = Reference count

The bound (b) specifies the limit on array index values, the virtual plane representation (vp) points to a plane in planar memory, and an optional reference count field can be associated with each plane so that they can be discarded when no longer accessible by the program, or updated destructively if only a single reference to the plane exists. A detailed discussion of reference counting schemes is beyond the scope of this thesis but the interested reader is referred to [Aasa88] for a discussion of implementations of aggregates in functional languages, [Hughes87] for a discussion of cyclic reference counting, and to [Glaser88] for a comparison of the efficiency of various reference counting schemes for functional languages.

For the sake of simplicity the internal representation of basic planar values as pointers to descriptors will not be used in the remainder of the thesis. The PAM instructions abstract this "housekeeping" information and treats basic planar values in the same way as basic scalar values in order to present a symmetric view of the scalar and planar instructions.

4.3.3.2. Closures

Suspended evaluations need to be constructed in both scalar and planar combinator code streams. Scalar closures also need to contain scalar and planar code entry points. This is necessary since the argument to `imap` may be an ordinary lambda-expression as in the following expression:

```
imap (λj.λx. + x 3) ar
```

Closures are constructed for lambda-expressions and in this example the planar entry point will be used. Once in the planar code stream only the planar entry points of combinators and suspended evaluations will be used (i.e. data-parallel renditions of functions may only call the data-parallel renditions of other functions). Therefore, closures constructed in the planar code stream do not require a scalar entry point. The scalar code stream is eventually resumed when the planar code stream returns to the original caller.

4.3.3.3. Algebraic Data-types

Algebraic data-types (ADTs) consist of closures whose code section causes the discriminator to be returned to the caller. The discriminator is a basic value stored in the closure along with pointers to the constructor arguments.

For scalar data-types, the discriminator is an unboxed basic scalar value and the arguments may be other unboxed scalar values or pointers to other closures. For planar data-types, the discriminator is a planar value. The constructor arguments may be other planar values in planar memory or may point to other closures in scalar memory. The closure representing the planar ADT is located in the scalar heap (as all closures are), but the constructor arguments may alternatively point to locations in either scalar or planar memory corresponding to whether the arguments are other closures or basic planar values.

The distinction between planes and arrays can now be made clear. Whereas it is possible to declare an array of pairs, or lists, or any other data-type, planes only contain basic values. An array of lists is represented internally by PAM as a succession of closures representing `cons` cells. The first argument to `cons` is a planar value, whereas the second points to the next closure. The planar representation of arrays is essentially the array element type turned "inside-out." An array of lists is represented as a list of planes. An array of pairs as a pair of planes, etc.

4.3.4. Activity Masks and Conditional Execution

Data-parallel conditional statements are possible through the use of *conditional execution.* `PLANAR_OP` relies on an *activity mask*, a plane of boolean values of the same dimensions as the operands to the planar instruction. Each basic planar operation (addition, multiplication, etc.) is only carried out on the elements in the plane whose corresponding element in the mask is set to true. Planar logical operations can operate on this activity mask just like any other boolean planar operand. The compilation rules given in the next chapter explicitly manipulate the activity mask to implement parallel forms of conditional statements.

An activity mask (M) is part of PAM's machine state. All `PLANAR_OP` instructions and all assignment operations (`:=`) on planar operands are affected by this mask. Words in a plane whose corresponding element in M is set to false ignore subsequent `PLANAR_OP` and assignment instructions. The effect of the activity mask on the instruction:

```
w3 := PLANAR_OP (+) [w1, w2]
```

where `w1`, `w2`, and `w3` are virtual planar values is shown in fig. 4.7. The locations in the activity mask (*M*) which are set to false are "greyed-out." The addition operation is only carried out (and the result assigned to) where the activity mask is clear (i.e. set to true).

An *if-then-else* expression consists of a *predicate*, a *consequent*, and an *alternative*. The predicate is an expression returning true or false. In the planar instruction stream, this predicate returns a planar boolean value. Some of the elements in the plane may be set to true, others to false. It is therefore necessary to execute both the consequent and the alternative, but only those elements set to true must execute the consequent and those set to false the alternative.

If w is the planar result of evaluating the predicate expression and M is the activity mask register, then all instructions in the consequent are executed after setting the activity

Effect of Activity Mask on Planar Instructions

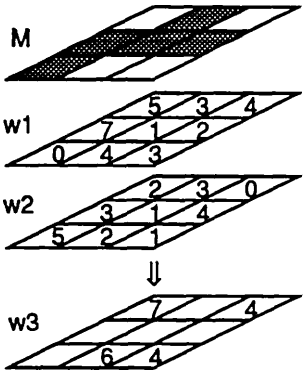


fig. 4.7

mask register to $w \wedge M$. Subsequently, the activity mask register is set to $\neg w \wedge M$ and all the instructions in the alternative are executed. This technique can be generalized to arbitrary conditional statements with multiple continuations.

Making sure that elements in the plane only execute the instructions appropriate to them is the source of much of the complexity in the compilation scheme for PAM. Since all continuations of a conditional statement need to be executed, a provision must be made to merge the partial results returned by each continuation into a single result.

In the case of the following expression:

```
λx. (= x 3) → 0; 1
```

If this function is applied as an argument to `imap`, then the result is a plane of the same size as `x`, where every location containing `3` in `x` is set to `0` in the result. All other locations are set to `1`. Both the consequent and the alternative are executed and the result of

the consequent is merged with the result of the alternative. This merging is accomplished by the use of the conditional assignment (:=) operation on planar values. The result value 0 is assigned to a plane using the activity mask appropriate for the consequent, whereas the result value 3 is assigned to the same plane using the activity mask appropriate for the alternative. No loss of referential transparency is involved since continuations of conditionals are mutually exclusive (i.e. a given element in a plane can only be active for a single continuation).

4.3.4.1. Contexts and Closures

Since planar instructions are affected by the contents of the activity mask register (*M*) the machine state affects the creation and execution of suspended evaluations in the planar code stream. Conventionally, a closure representing a suspended evaluation contains a pointer to a code address followed by pointers to the values of free variables occurring in the expression to be evaluated. In executing the scalar code stream of a combinator, this is sufficient. For the planar code stream, it is also necessary to save the state of the machine that affects the execution of the planar instructions. So far, the activity mask has already been shown to affect the basic planar instructions. The components of the machine state that affect the execution of planar instructions are collectively referred to as the *context.*

4.3.4.2. Saving Contexts

In executing suspended evaluations in the planar code stream, it is necessary to restore the context in which the closure was created. Closures in the planar code stream are henceforth referred to as *suspensions*, to differentiate them from the closures constructed in the scalar code stream. Suspensions must be executed in the context in which they were created, not in the context in which they are evaluated. The components of the context need to be stored in every suspension, in addition to the free variables of the suspended expression. Upon entering a suspension, the existing context is saved and the context stored in the suspension is made the current context. The remaining instructions in the suspension are executed, and just prior to returning, the saved context must be restored. This allows a suspension to be evaluated in the same context in which it was created. When the result is returned, the context is restored to what it was before the suspension was evaluated.

4.3.4.3. Conditional Statements

The general form of a `case`-statement is as follows:

```
case e0 in
        when c1 x11 ... x1n : e1
        when c2 x21 ... x2m : e2
        ...
        otherwise : eq
```

Each continuation contains an exclusive constructor which is used to select the continuation. For scalar code, the continuation whose constructor c_i matches the

discriminator returned by the evaluation of e_0 is entered. In the planar code stream, all continuations must be executed so an activity mask must be computed for each continuation. The evaluation of e_0 returns a basic planar value: the discriminator. This discriminator is compared with c_1. All locations in the plane which have M set to true and c_1 at the corresponding location in the discriminator execute the code for e_1.

The same process is repeated for each continuation. The exception is the `otherwise` continuation. In scalar mode, this continuation is taken if none of the others match the discriminator value. Since in the planar code stream all continuations must be executed, the `otherwise` continuation corresponds to all elements in the plane where corresponding elements in the discriminator did not match any of the c_i values. The activity mask for executing the otherwise expression (e_q) consists of the negation of the result of comparing each c_i value to the discriminator and computing the logical OR ($\vee$) of all comparisons. If the discriminator is denoted by d, then this activity mask is computed by the expression:

$$M \wedge \neg ((d = c_1) \vee (d = c_2) \vee \ldots \vee (d = c_{q-1}))$$

A consequence of executing every continuation of a `case`-statement is that it severely affects the way that algebraic data-types can be returned. Algebraic data-types are sum-types. That is, terms in an ADT are mutually exclusive and are differentiated by the use of a distinct tag i.e. the constructor. In the definition of `tree` the constructors `empty`, `leaf`, and `node` can be used to differentiate objects of type `tree`. A closure representing an ADT has the format: (*code*, *disc*, x_1, ..., x_n) where x_1, ..., x_n are constructor arguments. In a scalar machine, the fact that terms of an ADT are mutually exclusive can be used to optimize storage requirements. The maximum number of fields required for a closure representing any term of an ADT is equivalent to the term with the greatest number of constructor arguments. For closures representing the `empty` term, no additional fields are required. The first argument of a `leaf` is stored in field x_1. The first argument of a `node` is also stored in x_1, the remaining argument in x_2. The maximum number of fields (in addition to the code and discriminator field) required to represent any `tree` object is 2. Given an arbitrary closure representing a `tree` object, the status of the fields x_1, x_2 is completely defined by the value of the discriminator contained in the closure.

In the planar code stream the main complication is that as all continuations of a `case`-statements are executed which means that continuations that return an ADT no longer benefit from the mutual exclusion property exploited in scalar implementations. Each continuation may return a different term (or even the same term) of an ADT, which must somehow be combined into a single result. This presents a difficult problem whose solution is presented in §5.1.2.8.

4.3.4.4. Supporting Recursion

Functional languages typically have no iterative constructs, relying instead on recursion. In the planar code stream, the normal-order reduction scheme used in PAM already allows programmers to write algorithms containing unbounded recursion using infinite data-structures. This is due to the closure and suspension mechanism provided.

Bounded recursion, where some terminating condition is placed on the recursive expression can be handled readily because of the activity mask mechanism presented previously. The main problem is one of detecting the termination condition when this involves the use of expressions on planar operands. In the scalar code stream, an expression returns a discriminator which can be compared to desired values to decide whether to enter a recursive continuation or to stop and return some result. In the planar code stream, the difficulty lies in that some elements in the plane will meet the termination condition sooner than others. The activity mask mechanism already disables these locations from executing subsequent iterations. Recursion must stop once all elements in the activity mask are set to false. When this occurs, no further instructions will be obeyed. The answer therefore is to modify the SSTGM WHEN instruction so that in the planar code stream, a check is made on the activity mask of a given continuation to determine whether any elements in the mask are still able to carry out the instructions in the continuation (i.e. elements set to true in the activity mask). When all of the elements in the activity mask of a given continuation evaluate to false, then execution of this continuation can be avoided. This mechanism provides a way of terminating planar renditions of recursive functions. This mechanism is illustrated by way of an example in Chapter 6.

4.4. The Concrete Planar Abstract Machine: CPAM

PAM uses virtual planes. The size of planes in real machines is restricted because of the finite number of PEs. Virtual planes can be implemented using planes of fixed size by *tiling*. Multiple fixed-size planes, or *tiles*, are used to span a virtual plane of the desired size. The concrete planar abstract machine (CPAM) is a fixed-size planar architecture. CPAM is a general form of SIMD architecture but can also be used as a machine model for a MIMD implementation of PAM. The rest of this section is devoted to discussing the tiling mechanism used to implement the abstraction of virtual planes. The remainder of the thesis will continue to assume the virtual, PAM, target machine model for the sake of simplicity. The contents of this section serve only to aid the reader in developing an understanding of the low-level aspects of memory allocation in data-parallel architectures.

4.4.1. Tiling

Figure 4.8 shows how a 1-D array can be represented using a sequence of fixed-size (8×8, in this example) tiles (in the case of a 2-D target architecture).[2] Array elements are allocated starting from the top, left-most corner of the first tile, horizontally across and wrap around to succeeding rows. When the end of a tile is reached, allocation proceeds to another tile. The grey area represents the "un-used" area of the tile (e.g. if the array does not require all the available locations).

The virtual-plane component of an array descriptor is therefore refined to a list of tiles for each virtual plane. An activity mask is also permanently associated with each tile so that unused areas can be permanently masked out. It is not necessary to assign a mask to tiles that are fully utilized, but for simplicity it is assumed that a mask is associated with each tile. A tile descriptor is a 4-tuple:

(t, m, i, s) :
t = Tile
m = Mask
i = Tile number
s = Pointer to succeeding tile descriptor

Words in a tile are identified by their (*row*, *column*) location. Each tile used to implement a virtual plane is given a distinct number. The tile number and the bound on the array given in the array descriptor are sufficient to associate an array index with every location on every tile. A function, `indextoloc` is assumed, which returns a tile number and (*row*, *column*) location for a given bound and index value. Likewise, its inverse, `loctoindex` returns the array index for a given tile number and (*row*, *column*) location. These two functions can be used to convert between the array index and tile location values.

Figure 4.9 illustrates the tiling scheme for 2-D arrays using tiles of fixed size. The allocation of array elements proceeds horizontally across the first tile starting from the top, left-most corner. When the edge of the tile is reached, allocation proceeds into the next tile. Allocation "wraps around" when the limit on the horizontal dimension of the array is reached. The grey areas denote the unused areas of the tiles, masked out via the permanent activity mask associated with each tile in the tile descriptor.

This representation is demonstrated to illustrate how `imap` can provide an index for each array element as an additional argument to the function parameter. Expressly including this facility in the compilation scheme only serves to complicate matters so for clarity it will be assumed that the `map` operation is primitive instead (i.e. no index values are required for `map`).

---

[2]The term "tile" does not have any pre-defined geometrical dimensions here and is used only to describe a fixed-size analogue to it's virtual cousin, the plane.

Tiling 1-D Array into 8 ×8 Planes

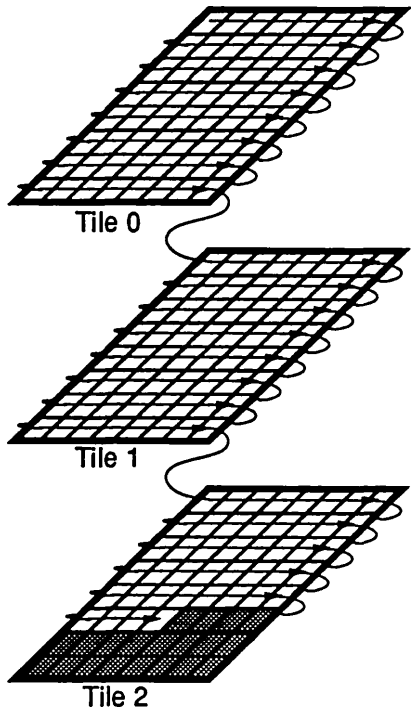


fig. 4.8

Tiling 2-D Array into 8 ×8 Planes

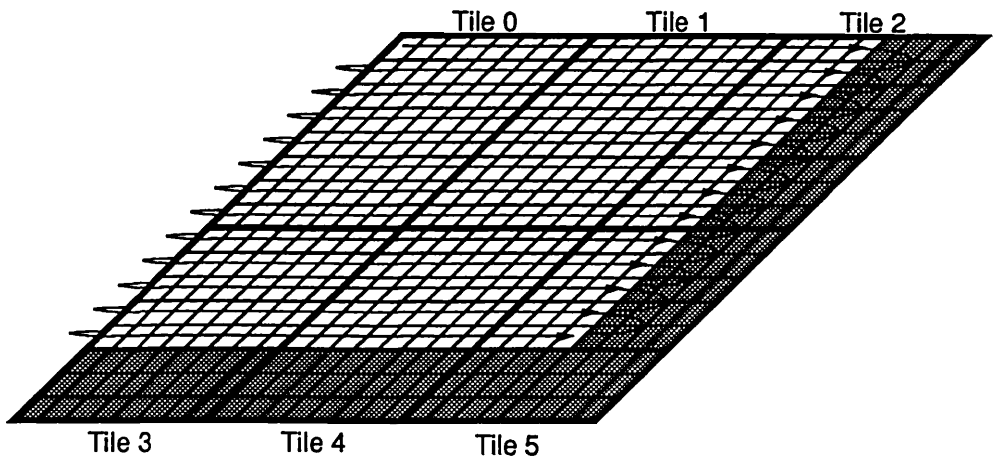


fig. 4.9

4.5. Data-Parallel Architecture Emulation

PAM provides an abstraction of SIMD architectures. SIMD architectures have been designed expressly for the exploitation of data-parallelism but the data-parallel model of computation can also be applied to MIMD machines. The *Virtual Systems Architecture*

(VSA) is an example of a high-level data-parallel model of computation which can be realized on both SIMD and MIMD systems [Jessho88]. The VSA model enables a style of programming based on the notions of *active data* and concurrent operations on abstract data-types. This model of computation presents a more general view of data-parallelism because it permits multiple (concurrent) flows of control.

In MIMD systems the central problem in exploiting data-parallelism lies in the distribution of data-structures. The ratio of communication-time to computation-time dictates the degree to which data should be distributed. A ratio of 1 indicates that data can be distributed extremely finely since the time to communicate a value is comparable to the time required to perform computation on it. A higher ratio indicates that data should be "clustered" so that a number of data elements reside on the same processor. The difficulty in clustering data centers on how temporal and spatial locality can best be exploited in the presence of latency.[3]

A distributed-memory MIMD emulation of PAM can exploit a readily available clustering mechanism by distributing entire tiles to processors. Each processor can be allocated a tile and tile size can be varied to take into account the performance of the processing elements versus the interconnection network. A process-parallel model of computation can be used for the scalar instructions of every process and a data-parallel model for planar instructions.

In the planar mode, the processor holding the array descriptor for a planar instruction can dispatch the instruction to all processors holding the tiles spanning the virtual plane. Each processor then applies the planar instruction to its local tile. In the planar mode instructions are sent to the data. To save on memory requirements, processors may be forced to synchronize on the completion of a planar instruction stream so that all tiles are in a consistent state (i.e. all elements have been provided a value). The resulting execution mechanism is analogous to the VSA or Valiant's proposed *Bulk Synchronous Parallel Machine* model [Valian89].

---

[3]In current MIMD inter-processor communication systems, communication latency is roughly one to two orders of magnitute greater than memory latency.

Chapter 5

5. Compilation to Planar Abstract Machine Code

In the previous chapter a compilation scheme for a simple scalar abstract machine was demonstrated and the necessary extensions to this sequential machine to exploit data-parallelism were discussed. In this chapter the instruction set of an abstract data-parallel machine is presented along with a compilation scheme to compile the extended functional language into abstract machine code. The instructions are given an operational semantics in terms of transitions on the machine state.

The abstract data-parallel machine is the *Planar Abstract Machine* (PAM). PAM has a dual instruction set corresponding to its scalar and planar memory modes. The problems involved in the compilation of a fully higher-order functional language are discussed.

Each function is compiled under two distinct compilation schemes to generate two renditions: one using scalar instructions, the other planar ones. The compiled code for each function therefore has two entry points, corresponding to the scalar and planar versions. Conditional statements, algebraic data-types, and recursive functions require special consideration under the planar compilation scheme. The compilation rules are illustrated through a series of small examples.

5.1. Compiling to PAM Code

PAM is an augmented form of SSTGM with a state denoted by the following tuple: $\langle C, H, S, E, N, X, R, M, B \rangle$. The additional components of the state are X, R, M, and B:

- An auxiliary stack for storing the context in which planar-instructions execute, X.
- A register to keep track of the number of arguments residing on the stack, R.
- A dedicated planar activity mask register, M.
- A register containing the bounds of created planar values, B.

The context stack is necessary to preserve the values of the activity mask and the bounds register (M and B) when executing suspended evaluations in the planar code stream. The R register keeps track of how many arguments are residing on the stack which is necessary for executing conditional statements in the planar code stream. The activity mask (M) determines which locations in a plane will execute the basic planar instructions, whereas the bounds register (B) sets the size of created basic planar values.

5.1.1. PAM Instructions

Many SSTGM instructions are encountered in both scalar and planar variants in PAM. In addition, some PAM instructions are new and are introduced to deal with special aspects of planar code execution. The only instructions which actually execute on planar values are `PLANAR_OP` and the planar form of assignment (`:=`). These two instructions and the `PLANAR_WHEN` instruction are the only ones that are affected by the state of the activity mask. An informal description of each instruction is given in fig. 5.1.

PAM Instructions

x	Variable, function, or constructor
u	Unboxed scalar basic value
w	Basic planar value
c	Constructor discriminator
p	Primitive operation
f	Function
t	Closure or Suspension

u := SCALAR_OP (p) [u_1, ..., u_n]	Apply the basic operation p to the scalar unboxed variables u_1, ..., u_n, storing the result in u.
w := PLANAR_OP (p) [w_1, ..., w_n]	Applythe basic planar operation p to all elements in the planes w_1, ..., w_n, subject to the corresponding location in the activity mask (M) being set to true and assign the elements in the active locations to w.
u := EVAL_SCALAR {*code*}	Execute the instructions in *code*, which will return an enumerated scalar basic value, to be stored in u.
w := EVAL_PLANAR {*code*}	Execute the instructions in *code*, which will return a basic planar value, to be stored in w.
f := COMBINATOR ({*scode*}, {*pcode*})	Define a new combinator in static code space with a scalar and planar code entry point, corresponding to the instructions in *scode* and *pcode* respectively.
t := CLOSURE [x_1, ..., x_n] ({*scode*}, {*pcode*})	Construct a closure (*scode*, *pcode*, x_1, ..., x_n) with both scalar and planar entry points in the scalar heap and point t to it.
t := SUSPENSION [x_1, ..., x_n] {*code*}	Construct a closure ($\emptyset$, *code*, M, B, x_1, ..., x_n) where *code* corresponds to the planar entry point and the scalar entry point is left empty (denoted by $\emptyset$).
NEW_CONTEXT	Save the context (M, B) on the context stack (X) and restore the old context contained in the suspension pointed to by N.
OLD_CONTEXT	Restore the context residing on the context stack.
RETURN_SCALAR u	Return the basic scalar value u to the element residing on the top of the stack.
RETURN_PLANAR w	Assign the basic planar value w to the location residing on the top of the stack.
RETURN_SCALAR_CONSTR c [x_1, ..., x_n]	Return the discriminator c to the element residing on the top of the stack, construct a closure (RETURN_SCALAR c, c, x_1, ..., x_n) in the heap and point N to it.
RETURN_PLANAR_CONSTR c [x_1, ..., x_n]	Convert c into a planar value (w) and assign it to the location residing on the top of the stack, construct a closure (RETURN_PLANAR w, w, x_1, ..., x_n) in the heap and point N to it.
ENTER_SCALAR x [x_1, ..., x_n]	Place x_1, ..., x_n on the stack and enter the scalar entry point of the closure or suspension pointed to by x.
ENTER_PLANAR x [x_1, ..., x_n]	Place x_1, ..., x_n on to the stack and enter the planar entry point of the closure pointed to by x.
POP x	Associate x with the top of the stack and decrement the stack pointer.

CASE {
alt_1,
...
alt_n
}
	Transfer execution to one of the continuations alt_1, ..., alt_n, if the corresponding constructor matches the discriminator returned by the evaluation of the predicate expression.

SCALAR_WHEN (c, u) [x_1, ..., x_n] {*code*}	Execute *code* if the discriminator u matches the constructor tag c. Constructor variables x_1, ..., x_n can be referenced as offsets from N.
PLANAR_WHEN [x_1, ... x_n] {*code*}	Execute *code* with planar constructor variables x_1, ..., x_n accessed as offsets from N.
SCALAR_OTHERWISE {*code*}	Execute *code* if no previous alternative is matched.
INIT_BOUND w	Set the bounds register (B) to the bounds of the planar value w.
MAKE_PLANAR u	Convert the basic scalar value u to a planar value with bounds B.

fig. 5.1

The instructions in fig. 5.1 are given an operational semantics in terms of transitions on the machine state in fig. 5.2.

State Transition Table for PAM Instructions

$\langle$(*u* := SCALAR_OP (*p*) [u_1, ..., u_n]::*code*)::*C*, *H*, *S*, *E*, *N*, *X*, *R*, *M*, *B*$\rangle$
$\Rightarrow$ $\langle$*code*::*C*, *H*, *S*, *E*[(*p* *E*[u_1], ..., *E*[u_n])/*u*], *N*, *X*, *R*, *M*, *B*$\rangle$

$\langle$(*w* := PLANAR_OP (*p*) [w_1, ..., w_n]::*code*)::*C*, *H*, *S*, *E*, *N*, *X*, *R*, *M*, *B*$\rangle$
$\Rightarrow$ $\langle$*code*::*C*, *H*, *S*, *E*[(@ *p* *E*[w_1], ..., *E*[w_n])/*w*], *N*, *X*, *R*, *M*, *B*$\rangle$

$\langle$(*u* := EVAL_SCALAR {$code_1$}::*code*)::*C*, *X*, *R*, *H*, *S*, *E*, *N*, *X*, *R*, *M*, *B*$\rangle$
$\Rightarrow$ $\langle$$code_1$::*code*::*C*, *H*, *u*::*E*::*S*, *E*, *N*, *X*, *R*, *M*, *B*$\rangle$

$\langle$(*w* := EVAL_PLANAR {$code_1$}::*code*)::*C*, *H*, *S*, *E*, *N*, *X*, *R*, *M*, *B*$\rangle$
$\Rightarrow$ $\langle$$code_1$::*code*::*C*, *H*, *w*::*R*::*E*::*S*, *E*, *N*, *X*, #*S*+3, *M*, *B*$\rangle$

$\langle$(t_1 := CLOSURE [x_1, ... x_n] ({*scode*}, {*pcode*})::*code*)::*C*, *H*, *S*, *E*, *N*, *X*, *R*, *M*, *B*$\rangle$
$\Rightarrow$ $\langle$*code*::*C*, t_2::*H*, *S*, *E*[t_2/t_1], *N*, *X*, *R*, *M*, *B*$\rangle$ t_2 = (*scode*, *pcode*, *E*[x_1], ..., *E*[x_n])

$\langle$(t_1 := SUSPENSION [x_1, ... x_n] {$code_1$}::*code*)::*C*, *H*, *S*, *E*, *N*, *X*, *R*, *M*, *B*$\rangle$
$\Rightarrow$ $\langle$*code*::*C*, t_2::*H*, *S*, *E*[t_2/t_1], *N*, *X*, *R*, *M*, *B*$\rangle$ t_2 = (∅, $code_1$, *M*, *B*, *E*[x_1], ..., *E*[x_n])

$\langle$(NEW_CONTEXT::*code*)::*C*, *H*, *S*, *E*, *N*, *X*, *R*, *M*, *B*$\rangle$
$\Rightarrow$ $\langle$*code*::*C*, *H*, *S*, *E*, *N*, *M*::*B*::*X*, *R*, *m*, *b*$\rangle$ *N* = (∅, *code*, *m*, *b*, ...)

$\langle$(OLD_CONTEXT::*code*)::*C*, *H*, *S*, *E*, *N*, *m*::*b*::*X*, *R*, *M*, *B*$\rangle$
$\Rightarrow$ $\langle$*code*::*C*, *H*, *S*, *E*, *N*, *X*, *R*, *m*, *b*$\rangle$

$\langle$(RETURN_SCALAR u_1::[])::*C*, *H*, u_2::*e*::*S*, *E*, *N*, *X*, *R*, *M*, *B*$\rangle$
$\Rightarrow$ $\langle$*C*, *H*, *S*, *e*[*E*[u_1]/u_2], *N*, *X*, *R*, *M*, *B*$\rangle$

$\langle$(RETURN_PLANAR w_1::[])::*C*, *H*, w_2::*r*::*e*::*S*, *E*, *N*, *X*, *R*, *M*, *B*$\rangle$
$\Rightarrow$ $\langle$*C*, *H*, *S*, *e*[*E*[w_1]/w_2], *N*, *X*, *r*, *M*, *B*$\rangle$

$\langle$(RETURN_SCALAR_CONSTR *c* [x_1, ..., x_n]::[])::*C*, *H*, *u*::*e*::*S*, *E*, *N*, *X*, *R*, *M*, *B*$\rangle$
$\Rightarrow$ $\langle$*C*, *t*::*H*, *S*, *e*[*c*/*u*], *t*, *X*, *R*, *M*, *B*$\rangle$
t = (RETURN_SCALAR *c*::[], ∅, *c*, *E*[x_1], ..., *E*[x_n])

$\langle$(RETURN_PLANAR_CONSTR *c* [x_1, ..., x_n]::[])::*C*, *H*, w_1::*r*::*e*::*S*, *E*, *N*, *X*, *R*, *M*, *B*$\rangle$
$\Rightarrow$ $\langle$*C*, *t*::*H*, *S*, e[w_2/w_1], *t*, *X*, *r*, *M*, *B*$\rangle$ w_2 = MAKE_PLANAR *c*,
t = (∅, RETURN_PLANAR w_2::[], w_2, *E*[x_1], ...,*E*[x_n])

$\langle$(ENTER_SCALAR *x* [x_1, ..., x_n]::[])::*C*, *H*, *S*, *E*, *N*, *X*, *R*, *M*, *B*$\rangle$
$\Rightarrow$ $\langle$*scode*::*C*, *H*, x_1::...::x_n::*S*, *E*, *t*, *X*, *R*, *M*, *B*$\rangle$ *t* = *E*[*x*] = (*scode*, *pcode*, y_1, ..., y_m)

$\langle$(ENTER_PLANAR x [x_1, ..., x_n]::[])::C, H, S, E, N, X, R, M, $B\rangle$
$\Rightarrow \langle pcode::C, H, x_1::...::x_n::S, \perp[E[y_1]/z_1, ..., E[y_m]/z_m], t, X, R, M, B\rangle$
$t = E[x] = (scode, pcode, m, b, y_1, ..., y_m)$

$\langle$(POP x_1::$code$)::C, H, x_2::S, E, N, X, R, M, $B\rangle$
$\Rightarrow \langle code::C, H, S, E[x_1/x_2], N, X, R, M, B\rangle$

$\langle$(CASE {alt_1, ..., alt_n}::[])::C, H, S, E, N, M, $B\rangle$
$\Rightarrow \langle [alt_1, ..., alt_n]::C, H, S, E, N, X, R, M, B\rangle$

$\langle$(SCALAR_WHEN (c, u) [...] {$code$}::$alts$)::C, H, S, E, N, X, R, M, $B\rangle$
$\Rightarrow \langle code::C, H, S, E, N, X, R, M, B\rangle$ if $(E[u] = c)$
$\Rightarrow \langle alts::C, H, S, E, N, X, R, M, B\rangle$ otherwise

$\langle$(PLANAR_WHEN [x_1, ..., x_n] {$code$}::$alts$)::C, H, y_1::...::y_q::S, E, N, X, R, M, $B\rangle$
$\Rightarrow \langle code::alts::C, H, y_1::...::y_q::w::R::E::y_1::...::y_q::S, E, N, X, R+(q+3), B\rangle$ $\#S = R$

$\langle$(SCALAR_OTHERWISE {$code$}::[])::C, H, S, E, N, X, R, M, $B\rangle$
$\Rightarrow \langle code::C, H, S, E, N, X, R, M, B\rangle$

$\langle$(INIT_BOUND w::$code$)::C, H, S, E, N, X, R, M, $B\rangle$
$\Rightarrow \langle code::C, H, S, E, N, X, R,$ MAKE_PLANAR true, $b\rangle$ $w = (b, vp, rc)$

fig. 5.2

In general, every closure and combinator must have a scalar and planar entry point. In certain cases only one entry point is required. For example, suspensions do not require a scalar entry point as they can only be entered via ENTER_PLANAR. Also functions taking or returning array-valued results cannot be used as arguments to `map` (as arrays may not be nested) hence these functions do not require a planar entry point either. The function argument to `map` may not contain any free variables which are arrays or use any of the array primitive operations. This is as a result of the restriction against nested parallelism described in Chapter 2. Where a given entry point is unused, this is denoted using the symbol $\varnothing$ which denotes a distinguished pointer value. It is assumed that attempting to enter an entry point set to $\varnothing$ results in a run-time error.

Several special symbols are used in the state transitions. The @ symbol takes a basic arithmetic/logical operation as a parameter followed by one or more planar values. The planar values in PAM are virtual planes. The @ operation symbolizes the application of a primitive operation to all the tiles representing the planar operands. The # symbol applied to the stack, S, is used to return a value denoting the depth of the stack. This is used in those instructions which need to store or retrieve the depth of the stack.

5.1.2. Compilation Scheme to Scalar and Planar Code

Compilation of the extended lambda-calculus into PAM code is illustrated in fig. 5.3. This compilation scheme is summarized in [Jouret91a]. Two versions of the $\mathcal{E}$, $\mathcal{C}$, and $\mathcal{B}$ scheme (corresponding to compilation rules for expressions, closures, and basic operations, respectively) are used. The scalar code stream is produced by the $\mathcal{ES}$, $\mathcal{CS}$, and $\mathcal{BS}$ translation rules. The planar code stream is produced by $\mathcal{EP}$, $\mathcal{CP}$, and $\mathcal{BP}$.

In Chapter 2, `imap` is primitive and `map` is derived in terms of `imap` but for the purpose of simplicity the compilation scheme presented below assumes that the `map` operation is primitive.

Compilation to PAM Code

k	Constant
x	Variable, function, or constructor
c	Constructor discriminator
p	Primitive operation
u	Unboxed basic scalar value
w	Basic planar value
f	Function
t	Closure or Suspension

$$
\begin{array}{lcl}
\mathcal{T}[\![f_1 = e_1 \ldots f_n = e_n; e]\!] & = & f_1 := \texttt{COMBINATOR}\ (\{\mathcal{ES}[\![e_1]\!]\ []\}, \{\mathcal{EP}[\![e_1]\!]\ []\}) \\
 & & \ldots \\
 & & f_n := \texttt{COMBINATOR}\ (\{\mathcal{ES}[\![e_n]\!]\ []\}, \{\mathcal{EP}[\![e_n]\!]\ []\}) \\
 & & \mathcal{ES}[\![e]\!]\ [] \\
 & & \\
\mathcal{ES}[\![k]\!]\ \sigma & = & u := k;\ \texttt{RETURN\_SCALAR}\ u \\
\mathcal{ES}[\![c]\!]\ \sigma & = & \texttt{RETURN\_SCALAR\_CONSTR}\ c\ \sigma \\
\mathcal{ES}[\![p\ e_1\ e_2]\!]\ [] & = & \mathcal{BS}[\![p\ e_1\ e_2]\!]\ u;\ \texttt{RETURN\_SCALAR}\ u \\
\mathcal{ES}[\![x]\!]\ \sigma & = & \texttt{ENTER\_SCALAR}\ x\ \sigma \\
\mathcal{ES}[\![\lambda x.e]\!]\ t::\sigma & = & x := t;\ \mathcal{ES}[\![e]\!]\ \sigma \\
\mathcal{ES}[\![\lambda x.e]\!]\ [] & = & \texttt{POP}\ x;\ \mathcal{ES}[\![e]\!]\ [] \\
\mathcal{ES}[\![e_1\ e_2]\!]\ \sigma & = & t := \mathcal{CS}[\![e_2]\!];\ \mathcal{ES}[\![e_1]\!]\ t::\sigma \\
\mathcal{ES}[\![\texttt{let}\ x_1 = e_1\ \texttt{in}\ e_2]\!]\ \sigma & = & x_1 := \mathcal{CS}[\![e_1]\!];\ \mathcal{ES}[\![e_2]\!]\ x_1::\sigma \\
\mathcal{ES}[\![\texttt{case}\ e\ \texttt{in} & & \\
\quad \texttt{when}\ c_1\ x_{11} \ldots x_{1i} : e_1 & & \\
\quad \texttt{when}\ c_2\ x_{21} \ldots x_{2j} : e_2 & & \\
\quad \ldots & & \\
\quad [\texttt{otherwise} : e_n]]\ \sigma & = & u := \texttt{EVAL\_SCALAR}\ \{\mathcal{ES}[\![e]\!]\ []\} \\
 & & \texttt{CASE}\ \{ \\
 & & \quad \texttt{SCALAR\_WHEN}\ (c_1, u)\ [x_{11} \ldots x_{1i}]\ \{\mathcal{ES}[[e]]\ \sigma\} \\
 & & \quad \texttt{SCALAR\_WHEN}\ (c_2, u)\ [x_{21} \ldots x_{2j}]\ \{\mathcal{ES}[[e]]\ \sigma\} \\
 & & \quad \ldots \\
 & & \quad [\texttt{SCALAR\_OTHERWISE}\ \{\mathcal{ES}[\![e]\!]\ \sigma\}] \\
 & & \} \\
 & & \\
\mathcal{CS}[\![k]\!] & = & k \\
\mathcal{CS}[\![x]\!] & = & x \\
\mathcal{CS}[\![e]\!] & = & \texttt{CLOSURE}\ \mathcal{FV}[\![e]\!]\ (\{\mathcal{ES}[\![e]\!]\ []\}, \{\mathcal{DP}[\![e]\!]\}) \\
 & & \\
\mathcal{BS}[\![k]\!]\ u & = & u := k \\
\mathcal{BS}[\![p\ e_1\ e_2]\!]\ u & = & \mathcal{BS}[\![e_1]\!]\ u_1 \\
 & & \mathcal{BS}[\![e_2]\!]\ u_2 \\
 & & u := \texttt{SCALAR\_OP}\ (p)\ [u_1, u_2] \\
\mathcal{BS}[\![e]\!]\ u & = & u := \texttt{EVAL\_SCALAR}\ \{\mathcal{ES}[\![e]\!]\ []\} \\
\mathcal{BS}[\![\texttt{map}\ e_1\ e_2]\!] & = & t := \mathcal{CP}\ [\![e_2]\!] \\
 & & w := \texttt{EVAL\_PLANAR}\ \{\texttt{ENTER\_PLANAR}\ t\ []\} \\
 & & \texttt{INIT\_BOUND}\ w \\
 & & \mathcal{EP}[\![e_1]\!]\ t::[]
\end{array}
$$

$\mathcal{CP}[\![k]\!]\ \sigma$ = w := MAKE_PLANAR k
RETURN_PLANAR w

$\mathcal{CP}[\![c]\!]\ \sigma$ = w := MAKE_PLANAR c
RETURN_PLANAR_CONSTR $w\ \sigma$

$\mathcal{CP}[\![p\ e_1\ e_2]\!]\ []$ = $\mathcal{BP}[\![p\ e_1\ e_2]\!]\ w$; RETURN_PLANAR w

$\mathcal{CP}[\![x]\!]\ \sigma$ = ENTER_PLANAR $x\ \sigma$

$\mathcal{CP}[\![\lambda x.e\ ]\!]\ t::\sigma$ = $x := t$; $\mathcal{CP}[\![e]\!]\ \sigma$

$\mathcal{CP}[\![\lambda x.e\ ]\!]\ []$ = POP x; $\mathcal{CP}[\![e]\!]\ []$

$\mathcal{CP}[\![e_1\ e_2]\!]\ \sigma$ = $t := \mathcal{CP}[\![e_2]\!]$; $\mathcal{CP}[\![e_1]\!]\ t::\sigma$

$\mathcal{CP}[\![\texttt{let}\ x_1 = e_1\ \texttt{in}\ e_2]\!]\ \sigma$ = $x_1 := \mathcal{CP}[\![e_1]\!]$; $\mathcal{CP}[\![e_2]\!]\ x_1::\sigma$

$\mathcal{CP}[$[case e in
when $c_1\ x_{11} \ldots x_{1i} : e_1$
when $c_2\ x_{21} \ldots x_{2j} : e_2$
...
[otherwise : e_n]]$\ \sigma$ =

w_0 := EVAL_PLANAR $\{\mathcal{CP}[\![e]\!]\ []\}$
$n_0 := N$; $m_0 := M$

w_1 := MAKE_PLANAR c_1
a_1 := PLANAR_OP $(=)$ $[w_0, w_1]$
m_1 := PLANAR_OP $(\wedge)$ $[M, a_1]$
$M := m_1$
v_1 := PLANAR_WHEN $[x_{11}, \ldots, x_{1i}]$ $\{\mathcal{CP}[\![e_1]\!]\ \sigma\}$
$n_1 := N$

$N := n_0$; $M := m_0$
w_2 := MAKE_PLANAR c_2
a_2 := PLANAR_OP $(=)$ $[w_0, w_2]$
m_2 := PLANAR_OP $(\wedge)$ $[M, a_2]$
$M := m_2$
v_2 := PLANAR_WHEN $[x_{21}, \ldots, x_{2j}]\{\mathcal{CP}[\![e_2]\!]\ \sigma\}$
$n_2 := N$
.
.
.

[$N := n_0$; $M := m_0$
w_n := PLANAR_OP $(\vee)$ $[a_1, a_2, \ldots a_{n-1}]$
a_n := PLANAR_OP $(\neg)$ $[w_n]$
m_n := PLANAR_OP $(\wedge)$ $[M, a_n]$
$M := m_n$
v_n := PLANAR_WHEN [] $\{\mathcal{CP}[\![e_n]\!]\ \sigma\}$
$n_n := N$
]

$N := n_0$; $M := m_0$
w_r := COMBINE $[v_1, v_2, \ldots, v_n]$ $[n_1, n_2, \ldots, n_n]$
$[m_1, m_2, \ldots, m_n]$
RETURN_PLANAR w_r

$\mathcal{CP}[\![k]\!]$ = MAKE_PLANAR k

$$
\begin{array}{lcl}
\mathcal{CP}[\![x]\!] & = & x \\
\mathcal{CP}[\![e]\!] & = & \texttt{SUSPENSION}\ \mathcal{FV}[\![e]\!]\ \{\mathcal{DP}[\![e]\!]\} \\
 & & \\
\mathcal{DP}[\![e]\!] & = & \texttt{NEW\_CONTEXT} \\
 & & w := \texttt{EVAL\_PLANAR}\ \{\mathcal{EP}[\![e]\!]\ []\} \\
 & & \texttt{OLD\_CONTEXT} \\
 & & \texttt{RETURN\_PLANAR}\ w \\
 & & \\
\mathcal{BP}[\![k]\!]\ w & = & w := \texttt{MAKE\_PLANAR}\ k \\
\mathcal{BP}[\![p\ e_1\ e_2]\!]\ w & = & \mathcal{BP}[\![e_1]\!]\ w_1 \\
 & & \mathcal{BP}[\![e_2]\!]\ w_2 \\
 & & w := \texttt{PLANAR\_OP}\ (p)\ [w_1, w_2] \\
\mathcal{BP}[\![e]\!]\ w & = & w := \texttt{EVAL\_PLANAR}\ \{\mathcal{EP}[\![e]\!]\ []\}
\end{array}
$$

fig. 5.3

Some aspects of the compilation of conditional statements for the planar code stream are worth mentioning here. The design of the PAM instructions reflects a desire to abstract the implementation-dependent aspects of compilation and to encapsulate a number of low-level tasks within a single, high-level instruction. For most of the compilation scheme, the machine state is indirectly manipulated through the high-level instructions.

A great part of PAM's complexity is due to the need to accommodate conditional statements in a transparent manner. Only in the rule for compiling conditional statements in the planar code stream are aspects of the machine state visible (e.g. the explicit manipulation of *N*, and *M*). This is the result of a conscious decision to explicitly illustrate the way conditional statements are supported in PAM. In §5.1.2.8 the way in which conditional statements are compiled is discussed in detail. The COMBINE operation is not a PAM instruction. It is a subroutine which when given a number of planar values, closures, and masks, combines them into a single result (see §5.1.2.8 & §5.1.2.9).

In the next few sections particular aspects of the compilation and the behaviour of PAM instructions are discussed in order to clarify the relation between the source language and the produced PAM-code. The compilation techniques which are unique to PAM include:

- The generation of double entry points for all combinators and closures.
- The mechanism to store the bound of created planar values.
- The compilation of `map`.
- The sharing of expressions between scalar and planar code streams.
- The evaluation and return mechanisms.

- The distinctions between suspended evaluations in the scalar and planar code streams.

- The maintenance of a consistent context for planar instructions.

- The execution of multiple continuations of conditional statements and the effect on the stack, the manipulation of the context, and the return of algebraic data-types.

5.1.2.1. Combinators

A program consists of a series of top-level function definitions which are compiled into combinators. Each combinator has two entry points and the code stream for each is generated by the $\mathcal{CS}$ and $\mathcal{CP}$ rules respectively. For example, the function:

```
constant = λx. 3
```

Yields the following PAM code:

```
constant := COMBINATOR ({
   POP x
   u1 := 3
   RETURN_SCALAR u1
}, {
   POP x
   w1 := MAKE_PLANAR 3
   RETURN_PLANAR w1
})
```

The two renditions of the function are clearly demonstrated. An instruction such as `ENTER_SCALAR constant` causes the first (scalar) entry point to be entered. In scalar mode, the argument to `constant` would necessarily be a scalar object (expression, basic value, or ADT).

5.1.2.2. Bounded Planar Arguments

In the previous function, `x` is never evaluated and the function returns the constant 3. The planar version of this function returns a plane all of whose elements contain the value 3. If `constant` is applied to an array via `map`, in the absence of additional information, it is unclear what the bounds of the returned planar constant should be. The solution is to store the bound of the array argument to `map` in the *B* register. The `MAKE_PLANAR` instruction creates planar objects (from scalar values) with the bound determined by *B*.

A related instruction, `INIT_BOUND w`, sets the bound register, *B*, to the bound of the basic planar value `w`. This instruction is always executed just prior to entering the planar code stream. It also resets the activity mask (*M*) so that initially, all elements are active (all elements in the mask are set to true):

(INIT_BOUND *w*::*code*)::*C*, *H*, *S*, *E*, *N*, *X*, *R*, *M*, *B*⟩
⇒ ⟨*code*::*C*, *H*, *S*, *E*, *N*, *X*, *R*, MAKE_PLANAR true, *b*⟩ $w = (b, vp, rc)$

The value used to initialize *B* is the bounds field (*b*) contained in the virtual plane descriptor representing the planar argument *w*. This ensures that subsequent planar code returns planar values of the dimensions given in *b*. This ensures that any array returned by `map` is of the same size.

5.1.2.3. Compiling `map`

In the expression `map constant` the planar entry point of `constant` is used and the argument `x` corresponds to the planar representation of the array (or an expression returning an array). The compiled code for the expression `map constant ar` is:

```
t1 := ar
w1 := EVAL_PLANAR { ENTER_PLANAR t1 [] }
INIT_BOUND w1
ENTER_PLANAR constant [t1]
```

The instruction `ENTER_PLANAR constant [t1]` uses the planar entry-point of `constant`. Note that `map` requires that its array argument be evaluated (this is done by `EVAL_PLANAR`). This is necessary so that the bounds register can be set to the dimensions of the array argument.

There is no source-language representation of `map` on arrays in PAM. The use of `map` is a syntactic convenience only. In our scheme `map` is a token which notifies the compiler that execution should switch from the scalar into the planar code stream.

The data-movement operations (`rotate`, `send`, and `fetch`) simply re-arrange data elements in the plane and can be implemented as calls to machine-specific communication routines. The compilation scheme therefore does not deal separately with the communication operations. They are simply translated into calls of the appropriate run-time library routines. This is why they do not appear in the compilation rules in fig. 5.3.

5.1.2.4. Sharing Expressions

Implementations of graph-reduction employ the use of *update markers* to benefit from the reduction of shared sub-expressions. If an expression is shared between two other expressions, then the reference to the shared expression is *marked* so that when it is reduced (yielding a new closure), the original expression is overwritten with an indirection to the (new) reduced form. In this way shared expressions are never re-evaluated and are only reduced once. A complication of PAM's dual scalar/planar code stream is that it is possible that expressions may be shared between the scalar and planar code streams. Consider the following expression:

```
map (λx. f y x) ar
```

where y is a free variable in the lambda-expression. This variable may point to an expression which was previously reduced in the scalar code stream. The variable y is shared between the scalar and planar code streams but it is impossible to use the scalar (reduced) form of y.[1] Indirections in PAM are just another type of closure and contain two entry points, a scalar and a planar one. As a result of reducing y in a scalar context the scalar entry point of y is updated to point to the new (reduced) form, but the planar entry point must continue to point to the original (un-reduced) form of y. Therefore, it is possible to retain the benefits of sharing only within a particular code-stream. The benefits of sharing do not extend across the code stream boundary.[2]

5.1.2.5. Evaluating & Returning

PAM has two variations of EVAL: EVAL_SCALAR and EVAL_PLANAR. Each is used to return a basic scalar and planar value respectively. In the scalar mode, basic values fit in a single word and are stored in an unboxed representation (i.e. not in a closure). A basic planar value is a pointer to a virtual plane descriptor.

The scalar form of EVAL is given as follows:

u := EVAL_SCALAR {*code*}

The instructions in *code* will cause a basic scalar value to be returned. Space for u is allocated on the stack and will be used by RETURN_SCALAR to return a basic value. The returned value is always assigned to the location on the top of the stack.[3]

The planar form of EVAL is similar except that a number of additional steps are carried out. This is evident from the state-transition definition:

$$\langle (w := \text{EVAL\_PLANAR } \{code_1\}::code)::C, H, S, E, N, X, R, M, B\rangle \Rightarrow \langle code_1::code::C, H, w::R::E::S, E, N, X, \#S+3, M, B\rangle$$

In PAM, arguments to functions are placed on the stack. In planar mode (because of the way conditional statements are treated, see §5.1.2.8) it is necessary to know (at any time) how many arguments are located on the top of the stack. The *R* register is provided for this purpose. At any point, the number of arguments on the stack is the difference between the location of the most recent return-address on the stack (placed there by an EVAL_PLANAR) and the current top of the stack. All locations between these two points are arguments to functions.

---

[1] A possible exception is when y is known to be a basic scalar value, in which case MAKE_PLANAR can be used to convert it to a planar value.

[2] An exception can be made in the case where a scalar expression has been reduced to a basic scalar value which can be turned into a planar value by MAKE_PLANAR which can be used to update the planar entry point.

[3] In a concrete implementation, the address to return to is stored in u until the value to be returned has been computed. To return control to the original caller, all that is required is to store the returned value in u and jump to the address that was stored there.

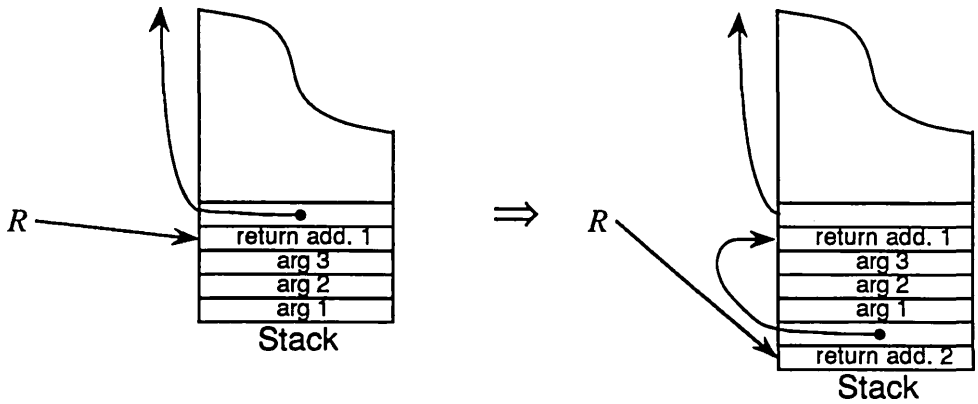
When an EVAL_PLANAR instruction is executed, it must store the previous value of R so that this can be restored upon executing a RETURN_PLANAR or RETURN_PLANAR_CONSTR. The state-transition for EVAL_PLANAR shows that the current contents of R are pushed onto the stack beneath the return-address w. The R register must then point to the new top of stack (#S+3). This constructs a linked-list of R values stored on the stack.[4] This is illustrated in fig. 5.4.

Control always returns to the instruction following an EVAL_PLANAR by the execution of a RETURN_PLANAR or RETURN_PLANAR_CONSTR instruction. Both of these instructions must restore the previous value of R lying under the return-address. The state-transition of the RETURN_PLANAR illustrates how R is restored:

$$\langle(\text{RETURN\_PLANAR } w_1::[])::C, H, w_2::r::e::S, E, N, X, R, M, B\rangle \Rightarrow \langle C, H, S, e[E[w_1]/w_2], N, X, r, M, B\rangle$$

The value to be returned, w_1, is used to update the location reserved for the returned result, w_2 ($e[E[w_1]/w_2]$). The previously-saved R register (denoted by the lower-case identifier r) is popped off the stack and restored.

Machine State During Execution of EVAL_PLANAR



(*i*) Before EVAL_PLANAR {*code*} (*ii*) Just before entering *code*

fig. 5.4

In the case of a RETURN_PLANAR_CONSTR instruction, the planar value returned to w is the discriminator. The N register is updated to point to the closure representing the ADT:

$$\langle(\text{RETURN\_PLANAR\_CONSTR } c\ [x_1, \ldots, x_n]::[])::C, H, w_1::r::e::S, E, N, X, R, M, B\rangle \Rightarrow \langle C, t::H, S, e[w_2/w_1], t, X, r, M, B\rangle \quad w_2 = \text{MAKE\_PLANAR } c,\ t = (\varnothing, \text{RETURN\_PLANAR } w_2::[], w_2, E[x_1], \ldots, E[x_n])$$

[4]This scheme is similar to the way update markers are stored on the stack in Peyton-Jones' original STGM.

The constructor *c* is turned into a planar value by MAKE_PLANAR. The closure representing the ADT is *t*. The *R* register is restored in the same way as before.

5.1.2.6. Closures & Suspensions

Compiling code for a dual instruction-type architecture like PAM is complicated by the fact that functions are first-class citizens in our functional language. In general, discovering which functions will be used as arguments to map (which causes planar versions to be required) is undecidable because the argument may only become known at run-time. Every function in a program may be required in both its scalar and planar renditions.[5] The compilation scheme needs to apply the scalar and planar translation rules to every function, producing two entry points for each combinator. For example, the function sqr shown below:

```
sqr : Int → Int

sqr = λn. (* n n)
```

when compiled produces the following PAM-code:

```
sqr := COMBINATOR ({
   POP n
   u1 := EVAL_SCALAR { ENTER_SCALAR n [] }
   u2 := SCALAR_OP (*) [u0, u1]
   RETURN_SCALAR u2
}, {
   POP n
   w1 := EVAL_PLANAR { ENTER_PLANAR n [] }
   w2 := PLANAR_OP (*) [w0, w1]
   RETURN_PLANAR w2
})
```

The ENTER_PLANAR n [] instruction causes the planar entry point of the closure for n to be entered, reducing it if it is an expression. The *N* register is changed to point to the closure for n, making its free variables (or constructor variables, in the case where the closure represents a planar ADT) available to subsequent instructions (as shown in the state-transition):

$$\langle(\text{ENTER\_PLANAR } x\ [x_1, \ldots, x_n]::[])::C, H, S, E, N, X, R, M, B\rangle$$
$$\Rightarrow \langle pcode::C, H, x_1::\ldots::x_n::S, \perp[E[y_1]/z_1, \ldots, E[y_m]/z_m], t, X, R, M, B\rangle$$
$$t = E[x] = (scode, pcode, m, b, y_1, \ldots, y_m)$$

This is denoted by executing *pcode* in a new environment ($\perp[]$) containing the free or constructor variables, represented by $z_1, \ldots, z_n$.

5.1.2.7. Context Stack

The PLANAR_OP and planar assignment (:=) instructions are affected by aspects of the machine state. In particular, the activity mask and the bounds register (referred to jointly as

[5] Subject to the restrictions mentioned earlier.

the *context*). Because of the mechanism of lazy evaluation, it is possible that the evaluation of an expression is deferred until some later point in the execution where the context has changed. To preserve the correct semantics, it is necessary to execute suspensions in the context in which they were *created*, not the one in which they are *evaluated*.

This requires that for an expression whose evaluation is being postponed, the context must be stored along with the free variables. Every suspension therefore contains the context at the time it was created, which can be re-instated when it is evaluated.

Note how suspensions are produced in the planar code stream:

$$\mathcal{CP}[\![e]\!] = \texttt{SUSPENSION}\ \mathcal{FV}[\![e]\!]\ \{\mathcal{DP}[\![e]\!]\}$$

$$\mathcal{DP}[\![e]\!] = \texttt{NEW\_CONTEXT}\ w := \texttt{EVAL\_PLANAR}\ \{\mathcal{CP}[\![e]\!]\ []\}\ \texttt{OLD\_CONTEXT}\ \texttt{RETURN\_PLANAR}\ w$$

The NEW_CONTEXT instruction takes the context stored in the current suspension and restores it. The existing values of M and B must be saved, however, so that they can be re-instated when returning from the suspension. A dedicated stack, the context-stack, exists for this purpose and is denoted by X. When the suspension is entered, the context at the time of creation is restored by NEW_CONTEXT and the current context is saved on X. The code for the expression is entered via an EVAL_PLANAR instruction. The use of EVAL_PLANAR to evaluate the expression forces control to return to the suspension. This is necessary because the context that existed prior to entering the suspension must be restored. This is done by the OLD_CONTEXT instruction which pulls the saved context off the context stack and re-instates it. The final RETURN_PLANAR instruction then returns to the original caller with the result of the suspension. The state transitions for the NEW_CONTEXT and OLD_CONTEXT instructions show how the context is saved and restored:

$$\langle(\text{NEW\_CONTEXT}::code)::C, H, S, E, N, X, R, M, B\rangle \Rightarrow \langle code::C, H, S, E, N, M::B::X, R, m, b\rangle \qquad N = (\varnothing, code, m, b, \ldots)$$

$$\langle(\text{OLD\_CONTEXT}::code)::C, H, S, E, N, m::b::X, R, M, B\rangle \Rightarrow \langle code::C, H, S, E, N, X, R, m, b\rangle$$

The closure pointed to by N must always have a nil scalar entry point (denoted by $\varnothing$) since the NEW_CONTEXT instruction only occurs within suspensions. The context saved in the suspension is denoted with lower-case symbols: m and b.

An additional stack (X) makes the implementation of an abstract machine more difficult. The argument stack (S) cannot readily be used to save the context because arguments may be residing on this stack which will be consumed as a result of entering the suspension. It is only "safe" to place values on the argument stack *under* the return address placed there by a previous EVAL_PLANAR. To store the context under this location

entails moving the arguments on the stack upwards every time a suspension is entered. A separate stack for contexts is therefore more efficient.

5.1.2.8. Executing Multiple Continuations

The problems introduced by conditional statements in the planar code stream have already been touched upon in the previous chapter. The main difference between the planar form of a conditional statement and a scalar one is that in planar mode, *all* continuations of a conditional statement are executed.

In scalar mode, a `case`-statement is compiled into a branch into the appropriate continuation. This continuation may pop additional arguments off the stack and create new temporary variables. In planar mode this requires that the stack be "restored" prior to executing each continuation (so that the same arguments are found on the stack by each continuation). As will be demonstrated, *M* and *N* need to be restored prior to each

Machine State During Execution of `PLANAR_WHEN`

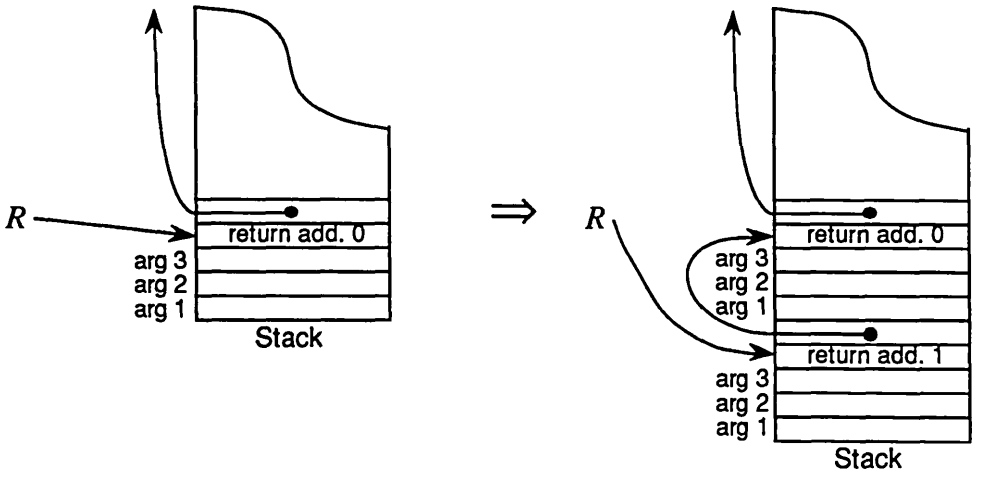


fig. 5.5

continuation as well.

In order to activate only those locations appropriate for each continuation, the activity mask will be modified for each continuation. After returning, the original activity mask must be restored.

Consider the following function:

```
alt = λb. case (b) in
   when true : f
   when false : g
```

Depending upon the value of `b`, `alt` returns the function `f` or `g`. Therefore, when `alt` is used in an expression, such as: `alt t x`, the arguments `t` and `x` are pushed on the stack before entering the code for `alt`. In planar mode, `t` is a planar value where some elements could be set to `false`, others to `true`. Both continuations of the `case`-statement

must therefore be executed. Branching to the first continuation and entering `f` will cause `x` to be popped off the stack. Branching to the second continuation, upon return from `f`, `x` will no longer be on the stack as it has been consumed by `f`. The `PLANAR_WHEN` instruction, therefore, needs to make a copy of all the arguments sitting on the stack which might be consumed by the continuation (see fig. 5.5). The number of arguments on the stack can be determined by inspecting R. R points to the most recent return-address. All the elements on the stack above the return-address are arguments which may be consumed by the continuation and therefore need to be preserved:

$$\langle(\text{PLANAR\_WHEN } [x_1, \ldots, x_n]\ \{code\}::alts)::C, H, y_1::\ldots::y_q::S, E, N, X, R, M, B\rangle$$
$$\Rightarrow \langle code::alts::C, H, y_1::\ldots::y_q::w::R::E::y_1::\ldots::y_q::S, E, N, X, R+(q+3), B\rangle \quad \#S = R$$

The `PLANAR_WHEN` instruction is similar to `EVAL_PLANAR` in that control returns to the instruction after `PLANAR_WHEN` upon execution of a `RETURN_PLANAR` or `RETURN_PLANAR_CONSTR` instruction. The difference lies in that the code section of a `PLANAR_WHEN` instruction is not executed if all the locations in the activity mask are set to false. The planar code produced for `alt` is documented below:

```
POP b
w0 := EVAL_PLANAR { ENTER_PLANAR b [] }        Evaluate Predicate Expression
n0 := N; m0 := M                                Save N and M

w1 := MAKE_PLANAR true                          Construct plane of all true
a0 := PLANAR_OP (=) [w0, w1]                    compare discriminator to all true
m1 := PLANAR_OP(∧) [M, a0]                      compute activity mask for true continuation
M := m1                                         Set activity mask
v0 := PLANAR_WHEN [] { ENTER_PLANAR f [] }      Evaluate f
n1 := N                                         Save value of returned N

N := n0; M := m0                                Restore value of initial N and M
w2 := MAKE_PLANAR false                         Construct plane of all false
a1 := PLANAR_OP (=) [w0, w2]                    compare discriminator to all false
m2 := PLANAR_OP (∧) [M, a1]                     compute activity mask for false continuation
M := m2                                         Set activity mask
v1 := PLANAR_WHEN [] { ENTER_PLANAR g [] }      Evaluate g
n2 := N                                         Save value of returned N

N := n0                                         Restore initial value of N
M := m0                                         Restore initial value of M
w3 := COMBINE [v1, v0] [n2, n1] [m2, m1]        Combine into 1 result
RETURN_PLANAR w3                                Return combined result
```

The activity mask for the `true` continuation corresponds to all those locations in the returned discriminator `w0` set to true and where the corresponding location in the original activity mask is set to true. This new activity mask is contained in `m1`. This needs to be saved for the final COMBINE operation because it needs to know which elements in `v0` actually contain results from the `true` continuation. Prior to executing the next continuation, the original *N* and *M* must be restored. This is because if `f` returns an ADT, then *N* will be pointing to the newly created closure representing an object of that data-type when control returns from `f`. *N* should be restored to its previous value so that free or constructor variables accessed in the continuations can remain accessible. The activity mask needs to be restored to its original state prior to entering the first continuation.

To see how COMBINE "unifies" returned values, consider the following example:

```
flip = λb. case (b) in
   when 0 : 1
   when 1 : 0
```

In this case, `b` is a basic planar integer where some elements may be set to `0`, others to `1`. Each continuation returns a constant planar value. Those elements in `b` which were set to `0` will be set to `1` in the result, those set to `1` will be set to `0`. Denoting the returned planar values as `w3` and `w4` and the activity mask for each continuation as `m3` and `m4`, the stack just prior to executing the COMBINE operation is illustrated in fig. 5.6. The activity masks are used by COMBINE to merge the two basic planar values into one by taking the values at the active locations from `w3` and `w4`. The result is a single planar value which is returned to the caller of `flip`.

Machine State During Execution of COMBINE

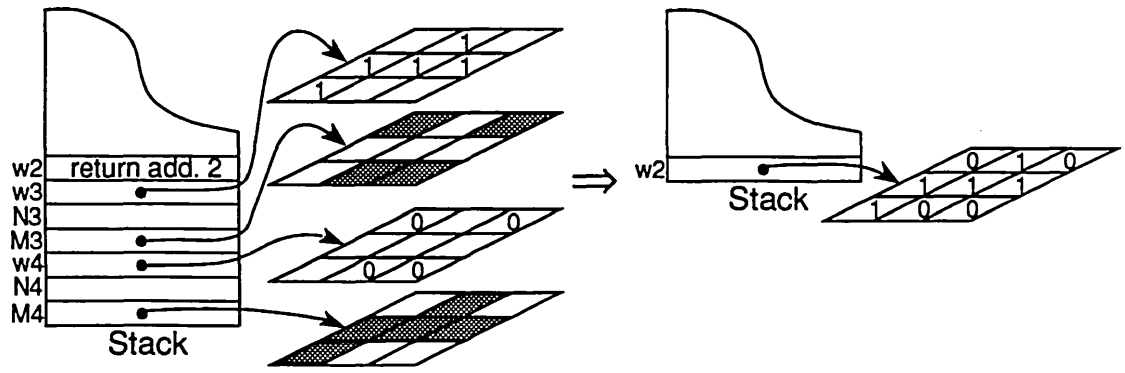


fig. 5.6

Continuations which return ADT results require *N* to be restored after each continuation. For example, consider the following tree-reversal function:

```
reverse = λt. case (t) of
   when empty : empty
   when leaf n : leaf n
   when node lt rt : node (reverse rt) (reverse lt)
```

Note that the evaluation of `t` causes `n`, `lt`, and `rt` to be available to the appropriate continuations. Unless *N* is reset to point to the closure returned by the predicate expression `t` (i.e. an object of type `tree`), access to these constructor variables will be lost after the first continuation is executed. Therefore, both *M* and *N* need to be restored to their original values before executing each continuation.

5.1.2.9. Returning Algebraic Data-types

A final complication arising from conditional statements in planar mode is that the implementation of ADTs needs to be modified. This was discussed briefly in the previous chapter. Conventionally, objects of sum-types are implemented as closures where sufficient fields are provided for the largest term of a sum-type. In that way, any returned term of a sum-type can be stored in the closure. This works because an expression can only return one term of a sum-type: the terms are necessarily mutually exclusive.

In PAM, this is no longer the case. Reconsider the `reverse` function in the previous section. The first continuation returns `empty`, the next a `leaf`, the last a `node`. Since all continuations must be executed, a single closure must allocate sufficient storage to store *all* the terms of a sum-type within a single closure. There is nothing exceptional about this: in the case of `reverse`, the result is an array of trees where the size of the trees at every element is allowed to vary (i.e. some elements may be nodes, others leaves, and others may be empty).

The solution is to forego the optimized storage scheme used by the scalar mode and to store sum-types as product-types. Sufficient space is allocated in the closure of a planar ADT for all arguments of all terms to coexist.

In other words, planar `tree` objects would use the following internal representation:

```
tree = empty × leaf × α × node × (tree α) × (tree α)
```

A planar object of type `tree` is stored as a product of its individual terms. The values of the discriminators are still mutually exclusive so the representation can be optimized to:

```
tree = (empty | leaf | node) × α × (tree α) × (tree α)
```

The discriminators are basic planar values and multiple constructors can co-exist in a single plane since it is known that no single location can possibly represent more than one term of `tree`. The `empty`, `leaf`, and `node` constructors are stored within a single discriminator as a basic planar value. The discriminator provides a way to identify which elements in the array correspond to the appropriate `tree` term.

The result is that `COMBINE` now has more work to do. In the case where continuations return ADTs, it must also combine the closures returned by each continuation into a single closure. This is why saving the value of *N* after evaluating each

continuation is necessary: if the continuation returns an ADT, N points to the returned closure.

For `reverse`, the COMBINE operation merges the returned planar constructor values into a single discriminator in the same way as ordinary basic planar values. The difficulty arises when COMBINE is asked to merge closures. A potential conflict exists: different continuations of a `case`-statement may return the same term of a sum-type. Consider the function `build`:

```
build = λn. case (< n 0) in
   when true : node (f n) empty
   when false : node (empty) (build (- n 5))
```

Two `node` terms are being returned, each with different constructor arguments! It is impossible to extend the product-type representation of ADTs to arbitrarily add fields to accomodate multiple `node` results. This is a substantial problem. There is a solution, however. It is possible to take advantage of the uniform representation of objects in the heap (i.e. suspended expressions are indistinguishable from ADTs) to return a result which is part data-object, part suspended expression.

When COMBINE tries to merge two ADTs which return the same term of a sum-type, it builds a closure as before, but points the constructor arguments to a special closure which when called will evaluate the two alternative argument expressions and then COMBINE their results! The code for this special closure (*merge*) is similar to the code created for `case`-statements. In fact, it has the following structure:

$$(\varnothing, \mathit{merge\_code}, m_1, e_1, m_2, e_2, \ldots, m_n, e_n)$$

Where $m_1, \ldots, m_n$ are mask values and $e_1, \ldots, e_n$ represent expressions whose results are to be combined. The code (*merge_code*) for this closure sets the activity mask M to m_1, then evaluates e_1, saving the returned N and M values, then does the same for each m_i, e_i. Finally, it executes a COMBINE operation to merge the values returned by evaluating $e_1, \ldots, e_n$. If the first `node` argument of the ADT returned by `build` is demanded, the special merge closure will be entered which causes each of its dependent expression to be evaluated to weak-head normal-form. The returned ADTs are then combined into a single closure which is returned to the original caller.

In fig. 5.7, the special merge closure is represented pictorially by ⊗ with arrows to the sub-expressions to be merged. Basic planar values on the stack are shown in boldface. Consequently, **node** refers to a plane all of whose elements are set to the constructor `node`. Initially, after the evaluation of the `case`-statement in `build`, two closures representing `node` terms have been constructed. The first closure contains `(f n)` as its first argument, the second `empty`. The evaluation of `(f n)` will return either `empty`, a `leaf`, or a `node`.

The expression `(f n)` is not evaluated at this time since it has not been demanded. Instead, COMBINE builds a merge closure (⊗) for the first argument to `node`:

$$(\varnothing, \mathit{merge}, \mathtt{M} \wedge (\mathtt{n} < \mathtt{0}), \mathtt{(f\ n)}, \mathtt{M} \wedge \neg\ (\mathtt{n} < \mathtt{0}), \mathtt{empty})$$

Which stores the activity mask appropriate for each expression. This is necessary so that COMBINE can fuse the returned values of each of the expressions into one. In fig. 5.7 the resulting closure shows the two node arguments pointing to merge closures which will cause the alternative sub-expressions to be evaluated and combined when required.

Machine State During Evaluation of Merge Closure

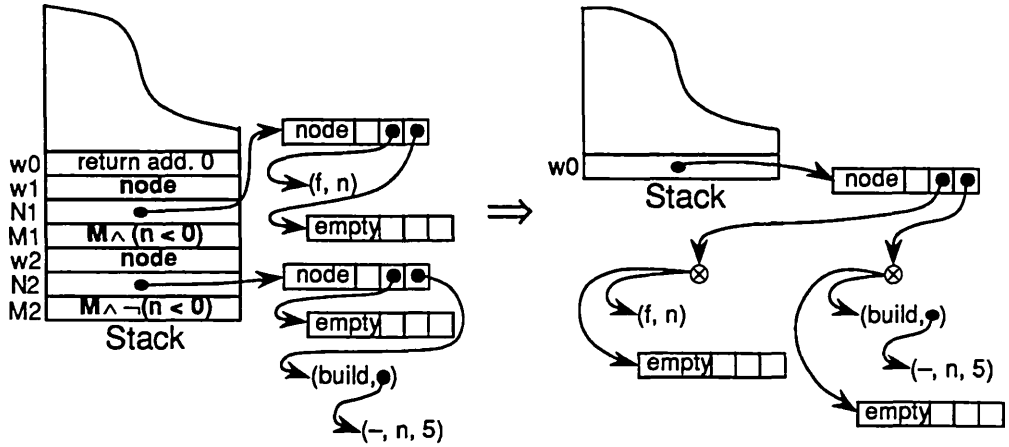


fig. 5.7

The COMBINE operation is quite general in that it not only combines basic values and ADTs but also inserts merge closures where necessary. All that is required is that COMBINE must have some way of knowing if the results sitting on the stack represent basic planar values or ADTs. If the continuations return basic planar values, then there is no need to inspect the saved values of *N* also sitting on the stack. This can be supported by the provision of a status bit which is set by RETURN_PLANAR_CONSTR and which can be detected (and cleared) by COMBINE.

Chapter 6

6. Code Generation Examples

Having presented a compilation scheme for our data-parallel functional language and an abstract machine instruction set to which programs are compiled, specific aspects of the compilation and execution can be illustrated using small program fragments to explain in detail how recursion, higher-order functions, algebraic data-types, and infinite data-structures are accomodated.

The execution of the resulting PAM code is discussed with respect to efficiency considerations. A space-time product efficiency metric is defined and used in discussing the efficiency of the code fragments presented in this chapter as well as some sample applications from Chapter 3. Inefficiency is attributed to machine characteristics, compilation, and algorithm choice. This leads to the consideration of a number of areas in which the compiled code could be improved by automatic and non-automatic optimizations.

6.1. Example Programs

In the next few sections a series of sample program fragments is considered. Each fragment has been selected to illustrate a specific aspect of the compilation and execution of PAM instructions.

6.1.1. Recursion

Existing functions defined on scalar values can then be applied in parallel to an array of data elements. These functions may include all the ordinary control structures available in the extended form of the lambda-calculus presented in Chapter 2. As an example, consider the function `gcd` below:

```
gcd = λu.λv.
   case (= v 0) in
      when true : u
      when false : gcd v (mod u v)
```

which returns the greatest common divisor of two numbers using Euclid's method. The function is defined on integer scalar values and can be applied to an array of integers as in the expression:

```
map (gcd 39) ar
```

The expression returns an array of the greatest common divisor of each element in the array `ar` and the constant `39`. The compilation system is responsible for generating a planar rendition of `(gcd 39)` so that `map` can then jump to it. Note that `gcd` is a recursive function and that the number of iterations is dependent upon the local value of each element in `ar`.

Compiling `gcd` yields the following PAM code:

```
gcd := COMBINATOR ({
   .
   .
   .
}, {
   POP u
   POP v
   w4 := EVAL_PLANAR {                                    check where v=0
      w5 := EVAL_PLANAR { ENTER_PLANAR v [] }
      w6 := MAKE_PLANAR 0
      w7 := PLANAR_OP (=) [w5, w6]
      RETURN_PLANAR w7
   }
   n0 := N
   m0 := M
   w8 := MAKE_PLANAR true
   a0 := PLANAR_OP (=) [w4, w8]
   m1 := PLANAR_OP (∧) [M, a0]
   M := m1
```

```
    v0 := PLANAR_WHEN [] { ENTER_PLANAR u [] }          where v=0 return u
    n1 := N
    N := n0
    M := m0
    w9 := MAKE_PLANAR false
    a1 := PLANAR_OP (=) [w4, w9]
    m2 := PLANAR_OP (∧) [M, a1]
    M := m2
    v1 := PLANAR_WHEN [] {                          where v≠0 call gcd again
       s0 := SUSPENSION [u, v] {
          NEW_CONTEXT
          w10 := EVAL_PLANAR {
             w11 := EVAL_PLANAR { ENTER_PLANAR u [] }
             w12 := EVAL_PLANAR { ENTER_PLANAR v [] }
             w13 := PLANAR_OP (mod) [w11, w12]
             RETURN_PLANAR w13
          }
          OLD_CONTEXT
          RETURN_PLANAR w10
       }
       ENTER_PLANAR gcd [v, s0]
    }
    n2 := N
    N := n0
    M := m0
    w14 := COMBINE [v1, v0] [n2, n1] [m2, m1]       combine 2 planes into 1
    RETURN_PLANAR w14
})
```

The scalar code section has been omitted since only the planar code is unique to PAM. Note how the evaluation of the expression `(mod u v)` has been delayed through the construction of a suspension (`s0`). The suspension will be entered if the value of the second argument to `gcd` is demanded. This example shows how lazyness is preserved in the generated code but also illustrates an example of where it is unnecessary: the evaluation of `gcd` will *always* require the second argument of `gcd` to be evaluated hence building a suspension to delay the evaluation of the argument is wasteful in both time and resources. Unnecessary suspensions such as these can be removed as an optimization step via strictness analysis. This will be discussed in Chapter 7.

The expression `map (gcd 39) ar` is compiled into the following PAM code:

```
s0 := ar
w0 := EVAL_PLANAR { ENTER_PLANAR s0 [] }
INIT_BOUND w0
w1 := MAKE_PLANAR 39
ENTER_PLANAR gcd [w1, s0]
```

Note how `s0`, an alias for `ar`, is passed to the planar version of `gcd` and not `w0`. In case a function is mapped onto an array whose elements are ADTs, `w0` would be the returned discriminator whereas `s0` would point to the closure representing the array (this was mentioned in the previous chapter). The constant `39` is converted into a planar value (`w1`) of the same size as `ar`. The *B* register has been set appropriately in the previous

instruction by INIT_BOUND so that MAKE_PLANAR knows how large the created plane should be. This new planar value can then be passed as an explicit argument to the planar entry point of gcd. A series of figures are used to illustrate how the planar version of gcd executes, showing how each of the locations in the the plane becomes de-activated in turn until no more active locations remain.

In fig. 6.1, the planar constant 39 is shown along with a plane of integer values corresponding to the u and v arguments of the planar rendition of gcd. The initial u and v planes are shown on the left and the arguments to the next iteration of gcd are shown on the right. None of the elements in the v plane contain 0 so the first PLANAR_WHEN code section will not be executed in the first iteration.

First Application of gcd

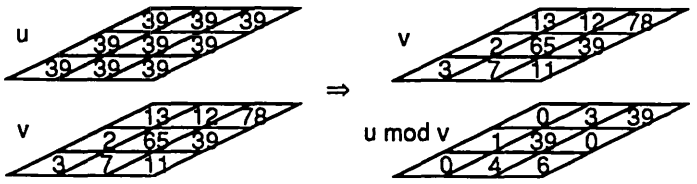


fig. 6.1

Second Application of gcd

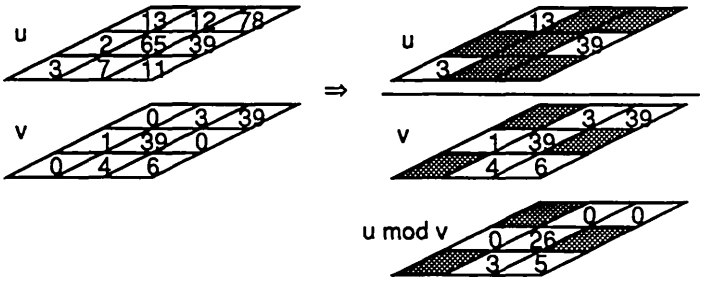


fig. 6.2

For the second iteration of gcd, the first argument is the original v and the second is u mod v. Note that 3 of the elements in the second argument are not set to 0. Hence, in fig. 6.2, the new u and v values are shown along with the two alternatives of each of the continuations of the conditional statement separated by a horizontal line. In this series of figures the activity mask is shown superimposed on the planar arguments.

Those locations in v which are set to 0 are rendered active in the first continuation, inactive in the second (inactive locations are "greyed-out"). The first continuation returns a planar value, whereas the second calls gcd for another iteration. The COMBINE operation at the end of the planar PAM code for gcd will combine the planar value returned by the first continuation with the value that will be returned by the call to gcd in the second. The

elements in the plane returned by the first continuation will form part of the final returned "answer". Those elements are rendered inactive for subsequent iterations.

A number of elements in the plane have already been inactivated at this point. The function `gcd` is then applied at the remaining active locations, as shown in fig. 6.3. Additional elements in `v` are set to `0`, so the first continuation returns the planar value `u` with those elements in `v` set to `0` activated. The activity mask for the second continuation is even more restricted as a result and only three locations remain active at this point.

For the fourth iteration of `gcd`, as shown in fig. 6.4, none of the elements in `v` are `0` so the first continuation is not executed. No planar value is returned by the first continuation. Instead, the second iteration performs another call to `gcd`.

Third Application of `gcd`

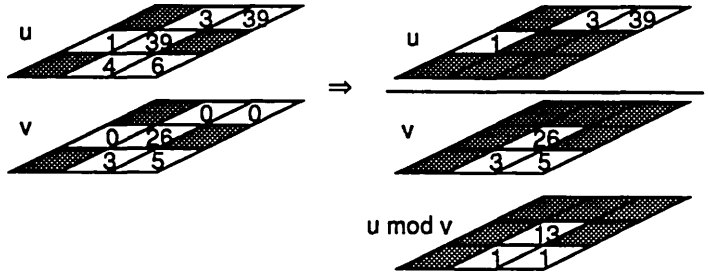


fig. 6.3

Fourth Application of `gcd`

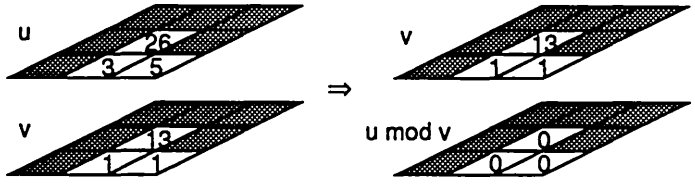


fig. 6.4

Finally, all remaining elements in `v` are set to `0` (fig. 6.5). At this point, the first continuation returns a planar value with the remaining three values. For the second continuation, all locations are now inactive so this continuation is not executed. The iteration is therefore terminated and execution continues to all the remaining `COMBINE` operations which merges all the intermediate values at their active locations to yield a single planar value.

This process is illustrated in fig. 6.6 where the planar values returned by the first continuation of each iteration are shown. Each intermediate result is then combined to yield a single planar value.

This example clearly illustrates the following aspects of compilation and execution:

- The generation of constant planar values.
- The compilation of partially-applied function argument to `map`.
- The automatic manipulation of activity mask by the compilation to permit use of arbitrary control structures including general recursion.
- The separate evaluation of all continuations of conditional statements followed by automatic combination of partial results to yield a single planar result.

Final Application of `gcd`

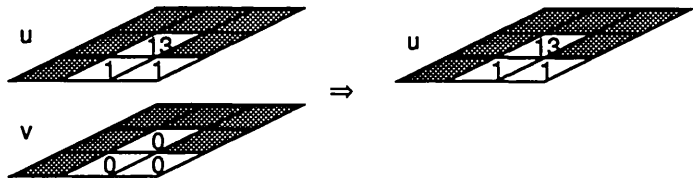


fig. 6.5

Combining Final Result of `gcd`

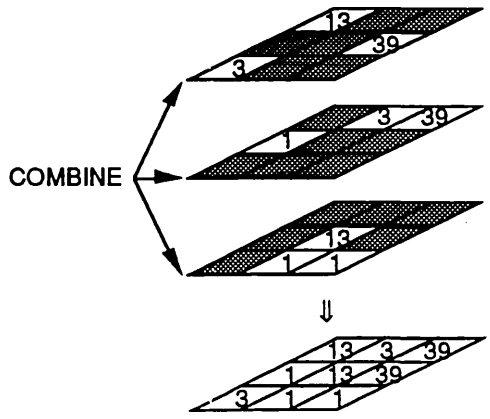


fig. 6.6

6.1.2. Higher-Order Functions

Higher-order functions can be defined in our language to yield powerful operations which are parameterized by other functions. The fact that functions are first-class objects in our language means that arrays may contain elements which are functions.

An array of functions can be a useful way of specifying computation. Consider the digital logic circuit simulation of Chapter 3, where an array whose elements are irrefutable

constructors (i.e. constructors without arguments) was used to selectively apply the AND function, OR function, or the FST function:

```
act = λg.λ(x, y).
   case g in
      when andgate : AND x y
      when orgate : OR x y
      when fst : FST x y

cycle = λin1.λin2.λgr.λrr.
   let xr = map hd (fetch in1 rr) in
   let yr = map hd (fetch in2 rr) in
      map2 act gr (zip xr yr)
```

This manner of specifying the computation is unnatural because at each iteration in `cycle`, the elements in `gr` are repeatedly tested to see what type of gate each element corresponds to before applying the appropriate function. It is more natural to consider assigning the functions AND, OR, and FST directly to the appropriate locations in the array `gr`, at which point `cycle` merely becomes:

```
cycle = λin1.λin2.λgr.λrr.
   let xr = map hd (fetch in1 rr) in
   let yr = map hd (fetch in2 rr) in
      map2 apply gr (zip xr yr)
```

Where `apply` is defined as `λf.λ(x, y). f x y`. The array `gr`, instead of acting as a "passive" array of constructors which are used to discriminate for the functions which represent the logic gates, becomes an "active" data-structure, with elements corresponding directly to the functions themselves. Before this new definition of `cycle` can be used, the original array `gr` must be converted into an array of function-valued elements. This can be done as follows:

```
convert = λgr. map (λg.
   case g in
      when andgate : AND
      when orgate : OR
      when fst : FST) gr
```

The new program then contains a call to:

```
cycle in1 in2 (convert gr) rr
```

The converted array becomes an argument to `cycle`. The compiled code for `convert` is given below:

```
convert := COMBINATOR ({
   POP gr
   s0 := gr
   w0 := EVAL_PLANAR { ENTER_PLANAR s0 [] }        evaluate gr
   INIT_BOUND w0
   g := s0
   w1 := EVAL_PLANAR { ENTER_PLANAR g [] }
```

```
    n0 := N
    m0 := M
    w2 := MAKE_PLANAR andgate
    a0 := PLANAR_OP (=) [w1, w2]
    m1 := PLANAR_OP (&) [M, a0]
    M := m1
    v0 := PLANAR_WHEN [] { ENTER_PLANAR AND [] }   gr=andgate call AND
    n1 := N
    N := n0
    M := m0
    w3 := MAKE_PLANAR orgate
    a1 := PLANAR_OP (=) [w1, w3]
    m2 := PLANAR_OP (&) [M, a1]
    M := m2
    v1 := PLANAR_WHEN [] { ENTER_PLANAR OR [] }    gr=orgate call OR
    n2 := N
    N := n0
    M := m0
    w4 := MAKE_PLANAR fst
    a2 := PLANAR_OP (=) [w1, w4]
    m3 := PLANAR_OP (&) [M, a2]
    M := m3
    v2 := PLANAR_WHEN [] { ENTER_PLANAR FST [] }    gr=fst call FST
    n3 := N
    N := n0
    M := m0
    w5 := COMBINE [v2, v1, v0] [n3, n2, n1] [m3, m2, m1]   combine all
    RETURN_PLANAR w5
}, {})
```

The calls to AND, OR, and FST can readily be recognized. A suspension will be built for the partially applied form of `convert` in the call to `cycle`. A function is not entered until all of its arguments are available so as a result, this form of `convert` is not any more efficient than the previous version which tested the elements in `gr` explicitly in each iteration to determine which constructor they contained. The new version conceptually returns an "array of functions" but what actually happens is that this is represented internally as a partially-applied planar function, waiting for additional planar arguments before it is executed! It is no longer necessary to test the constructor values of each of the elements in `gr` explicitly but this is what the new compiled version of `convert` does anyway. The original array `gr` is permanently bound up as an argument to the partially-applied call to `convert`. Once the remaining argument is made available, a call to this partially-applied function causes it to test the elements in `gr` by the sequence of PLANAR_WHEN instructions.

An array of functions is in fact a suspension representing a partially-applied planar function. The mechanism for suspensions is already available in PAM so this provides support for arrays with "function-valued elements" without additional complexity. This example illustrates the following compilation aspects:

- Arrays with function-valued elements are compiled into partially-applied planar functions.

- Using arrays of function-valued elements provides the abstraction of "active" data-structures but is in fact no more efficient than explicitly using constructors and testing them explicitly since testing is performed implicitly.

6.1.3. Algebraic Data-Types

In addition to supporting higher-order functions, one of the most surprising aspects of the compilation scheme is that it is possible to define arrays whose elements are recursive algebraic data-types. This means that programs can make use of arrays of lists, trees, or any user-defined recursive structures. Furthermore, because our language is fully lazy, it becomes possible to use arrays of *infinite* data-structures. Functions defined on conventional recursive data-types can simply be mapped across an array whose elements are of this type.

Infinite data-structures are useful when it is unknown how "much" of the data-structure needs to be computed to yield a result. For example, consider the Newton-Raphson root-finding method, where successive approximations to the root x_n of a function F is computed by the following recurrence equation:

$$x_{n+1} = x_n + \frac{F(x_n)}{F'(x_n)}$$

Where F' is the derivative of the function F. The function `newton` generates an infinite list of approximations to the root given some initial approximation:

```
newton = λx.λf.λf'.
   let y = (+ x (/ (f x) (f' x))) in
       cons y (newton y f f')
```

In order to yield a result in a finite amount of time, the initial segment of the list is returned up to the point where successive approximations to the root differ only by some sufficiently small quantity, ε. The function `enough` returns this finite segment:

```
enough = λxs.
   let x1 = hd xs in
   let ys = tl xs in
   let x2 = hd ys in
       case (< (abs (- x1 x2)) ε) in
          when true : cons x1 (cons x2 nil)
          when false : cons x1 (enough ys)
```

When applied to an expression which returns an infinite list of approximations to roots of functions, `enough` will return the initial segment until successive approximations differ by less than ε. Applying `enough` and `newton` in parallel via `map` is straightforward:

```
map (λa.enough (newton a f f')) ar
```

The normal-order evaluation strategy forces `newton` to be evaluated only as often as necessary at each element in the array until successive approximations are sufficiently close. The result is an array of list values. For example for the function F:

$$F(x) = x^3 + 3x^2 + 2x - 0.375$$

$$\frac{\partial}{\partial x}F = F'(x) = 3x^2 + 6x + 2$$

Applied to an initial array of approximations ≪-10, 0.7, 3≫ with $\varepsilon = 1 \times 10^{-10}$ returns the array of lists consisting of the approximations to the roots from these initial guesses:

```
≪ [-6.352555248618780,   [0.560273972602740,   [2.48863636363636, ≫
   -3.929788234853880,    0.511934288054993,    2.23452701874134,
   -2.327014835060680,    0.500736717942147,    2.15839334943551,
   -1.275233634337180,    0.500003224832608,    2.15144407473562,
   -0.595405862130317,    0.500000000062395,    2.15138782253751,
   -0.167372458551033,    0.500000000000000]    2.15138781886600]
    0.091177746945457,
    0.237893031723960,
    0.312689017246702,
    0.342345210190650,
    0.348352570235485,
    0.348611700208770,
    0.348612181132346,
    0.348612181134003]
```

Which are the three roots of the function F. By looking at the PAM code generated for `newton` and `enough` it is possible to see how the data-parallel application of these two functions preserves lazyness.

The compiled code for the function `newton` is as follows:

```
newton := COMBINATOR ({
   .
   .
   .
}, {
   POP x
   POP f
   POP f'
   y := SUSPENSION [x, f, f'] {
      NEW_CONTEXT
      w7 := EVAL_PLANAR {                    y = (-x (/ (f x) (f' x)))
```

```
            w8 := EVAL_PLANAR { ENTER_PLANAR x [] }
            w9 := EVAL_PLANAR { ENTER_PLANAR f [x] }
            w10 := EVAL_PLANAR { ENTER_PLANAR f' [x] }
            w11 := PLANAR_OP (/) [w9, w10]
            w12 := PLANAR_OP (-) [w8, w11]
            RETURN_PLANAR w12
         }
         OLD_CONTEXT
         RETURN_PLANAR w7
      }
      s0 := SUSPENSION [y, f, f'] {                    recursive call to newton
         NEW_CONTEXT
         w13 := EVAL_PLANAR { ENTER_PLANAR newton [y, f, f'] }
         OLD_CONTEXT
         RETURN_PLANAR w13
      }
      w14 := MAKE_PLANAR cons
      RETURN_PLANAR_CONSTR w14 [y, s0]                          return cons
   })
```

Constructors are lazy so suspensions are built for both arguments to `cons`. The first argument is the expression `(- x (/ (f x) (f' x)))`. A large suspension, `y`, is built for this expression. Similarly, a suspension `s0` is built for the recursive call to `newton`. It is the laziness of the constructor which allows this function to be infinitely recursive (note the lack of a base case in `newton`). Termination is the duty of the function `enough`, which calls `newton`. The compiled code for `enough` is given below:

```
enough := COMBINATOR ({
   .
   .
   .
}, {
   POP xs
   x1 := SUSPENSION [xs] {                                    hd xs
      NEW_CONTEXT
      w10 := EVAL_PLANAR { ENTER_PLANAR hd [xs] }
      OLD_CONTEXT
      RETURN_PLANAR w10
   }
   ys := SUSPENSION [xs] {                                    tl xs
      NEW_CONTEXT
      w11 := EVAL_PLANAR { ENTER_PLANAR tl [xs] }
      OLD_CONTEXT
      RETURN_PLANAR w11
   }
   x2 := SUSPENSION [ys] {                                    hd ys
      NEW_CONTEXT
      w12 := EVAL_PLANAR { ENTER_PLANAR hd [ys] }
      OLD_CONTEXT
      RETURN_PLANAR w12
   }
   w13 := EVAL_PLANAR {                          (< (abs (- x1 x2)) ε)
      w14 := EVAL_PLANAR { ENTER_PLANAR x1 [] }
      w15 := EVAL_PLANAR { ENTER_PLANAR x2 [] }
      w16 := PLANAR_OP (-) [w14, w15]
      w17 := PLANAR_OP (abs) [w16]
```

```
        w18 := EVAL_PLANAR { ENTER_PLANAR ε [] }
        w19 := PLANAR_OP (<) [w17, w18]
        RETURN_PLANAR w19
    }
    n0 := N
    m0 := M
    w20 := MAKE_PLANAR true
    a0 := PLANAR_OP (=) [w13, w20]
    m1 := PLANAR_OP (&) [M, a0]
    M := m1
    v0 := PLANAR_WHEN [] {                    less than ε so cons x1 (cons x2 nil)
        s1 := SUSPENSION [x2] {
            NEW_CONTEXT
            w21 := EVAL_PLANAR {
                s2 := SUSPENSION [] {
                    NEW_CONTEXT
                    w22 := EVAL_PLANAR {
                        w23 := MAKE_PLANAR nil
                        RETURN_PLANAR_CONSTR w23 []
                    }
                    OLD_CONTEXT
                    RETURN_PLANAR w22
                }
                w24 := MAKE_PLANAR cons
                RETURN_PLANAR_CONSTR w24 [x2, s2]
            }
            OLD_CONTEXT
            RETURN_PLANAR w21
        }
        w25 := MAKE_PLANAR cons
        RETURN_PLANAR_CONSTR w25 [x1, s1]
    }
    n1 := N
    N := n0
    M := m0
    w26 := MAKE_PLANAR false
    a1 := PLANAR_OP (=) [w13, w26]
    m2 := PLANAR_OP (&) [M, a1]
    M := m2
    v1 := PLANAR_WHEN [] {                    greater than ε so cons x1 (enough ys)
        s3 := SUSPENSION [ys] {
            NEW_CONTEXT
            w27 := EVAL_PLANAR { ENTER_PLANAR enough [ys] }
            OLD_CONTEXT
            RETURN_PLANAR w27
        }
        w28 := MAKE_PLANAR cons
        RETURN_PLANAR_CONSTR w28 [x1, s3]
    }
    n2 := N
    N := n0
    M := m0
    w29 := COMBINE [v1, v0] [n2, n1] [m2, m1]
    RETURN_PLANAR w29
})
```

The `enough` function forces the evaluation of `newton` by trying to evaluate successive values of the input list. The top-level call consists of the expression:

```
map (λa. enough (newton a f f')) ar
```

Which results in the following PAM-code:

```
s0 := ar
w0 := EVAL_PLANAR { ENTER_PLANAR s0 [] }
INIT_BOUND w0
a := s0
s1 := SUSPENSION [a, f, f'] {
   NEW_CONTEXT
   w1 := EVAL_PLANAR { ENTER_PLANAR newton [a, f, f'] }
   OLD_CONTEXT
   RETURN_PLANAR w1
}
ENTER_PLANAR enough [s1]
```

The list-argument to `enough` consists of the list produced (on demand) by `newton`. There are two continuations in the conditional statement, the first corresponds to the base case of `enough` and the second to the recursive case. Assume that initially all elements in the array of lists are active. The call to `enough` forces the head of each list in the array to be evaluated by calling `newton`. An array of list-elements is represented internally as a list of planes. If the none of the initial approximations to the root is sufficiently accurate (i.e. differs from the initial guess by more than the value of ε), then the approximations are added to the head of each list. The next approximation is then demanded by `enough`. When an approximation is sufficiently good, then the first continuation of the conditional statement in `enough` is taken. In a process similar to that shown for the execution of `gcd` in §6.1.1, these locations are now rendered inactive with regard to the second continuation, the recursive call to `enough`. In other words, the lists at these elements are terminated and the planar ADT returned in `v0` will be combined with the planar ADT returned in `v1` once the second continuation returns. The activity mask mechanism excludes those elements from partaking in any further calls to `enough`. Once the entire activity mask is set to false, i.e. all elements have reached sufficiently good approximations, the `COMBINE` operation fuses all the partial results into one common planar ADT representing the array of lists.

In this example the following points were illustrated:

- The effect of lazy evaluation on computation involving arrays of infinite algebraic data-types.

- The way in which infinitely recursive functions are "driven" to yield finite results.

This example is slightly contrived in that it is unlikely that the list of successive approximations to the root is a useful result: the final approximation is probably good enough. Nevertheless, it is a simple example which shows that arrays of algebraic data-types can be supported with full generality. However, support for conditional statements and algebraic data-types is costly in PAM. The question of efficiency is addressed next.

6.2. Efficiency Considerations

So far, the efficiency of the primitive operations and the way in which functions are compiled has only briefly been touched upon. What metric should be used to quantify efficiency? Ideally, all the PEs in a parallel machine should be engaged in useful work all of the time. This suggests an efficiency metric containing the space-time product (i.e. the number of elements in an array multiplied by the number of time units). That is a measure of the total resources *used*, so by computing the sum of all active elements in an array for each time unit t (for a total time of n units), the efficiency becomes:

$$E = \frac{\sum_{t=1}^{n} \#active\ locations}{\#locations \cdot n}$$

This gives a measure of how efficiently the resources of space and time are actually used. The maximum efficiency is 1. On synchronous architectures such as SIMD machines efficiency calculations are easy because there is only a single stream of instructions and all PEs must either execute the instructions broadcast or render themselves inactive.

A metric for efficiency allows the quality of different programs which exploit data-parallelism to be analyzed. Efficiency becomes an aspect of the complexity of abstract machine instructions, limitations of the underlying architecture, the compilation scheme, and the particular algorithm used in a program.

Ideally, the source of most ineffiency should be due to either architecture restrictions or algorithm choice. In that way, the quest for greater efficiency can lead the programmer to either a different machine architecture or a different kind of algorithm. The quest for transparency in the data-parallel operations guarantees that the primitive operations are efficient. Each of the primitive operations (with the exception of `fetch` and `send`) are constant-time operations. The efficiency of the `reduce` operation (for a constant-time function argument) on a SIMD machine can be derived from its definition by observing that $\lceil 1+lg(n) \rceil$ iterations of `rotate` and `map` are required (where n is the `bound` of the array). Therefore, the efficiency of `reduce` is:

$$E = \frac{\sum_{i=0}^{\lceil 1+lg(n) \rceil} \frac{n}{2^i}}{\lceil 1+lg(n) \rceil \cdot n} \approx \frac{2}{lg(n)}$$

This is as expected since half of the remaining active locations ($n/2^i$) are rendered "inactive" each iteration. The lack of conditional communication primitives means that locations are not rendered inactive in the sense of the `gcd` example in §6.1.1. In the case of `reduce` defined in terms of `scan`:

```
scan = λf.λar.
   let n = ceil (lg (+ 1 (bound ar))) in
   let (_, _, _, rr) = (spin ^ n)
      (λk.λj.λ(x, y). (> j k → f x y; x), 1, 0, ar) in
         rr
```

conditional communication is simulated by rotating the elements unconditionally and then leaving the "inactive" elements in place (e.g. `(> j k → f x y; x)`, where all elements whose index (`j`) is less than `k` remain unchanged). In the case of `reduce`, the number of inactive PEs can easily be determined because the conditional statement operates on the *indices* of array elements and not their values. In the more general case of conditional statements inactivating PEs on the basis of the *values* of array elements it may become impossible to give an equation for the efficiency of a data-parallel program fragment. Instead, graphical methods can be used to illustrate the efficiency for a sample input. For example, in the `gcd` example at the beginning of this chapter, the number of active locations per iteration of `gcd` can be shown, as in fig. 6.7. The grey areas correspond to inactive locations. Intuitively therefore, the amount of grey in such a diagram reflects the relative inefficiency of a program fragment. In this example, and all subsequent examples, it is assumed that the test of the activity mask performed by `PLANAR_WHEN` is a constant-time operation. In fact, as discussed in Chapter 5, this operation requires a reduction on the activity mask to discover whether any active elements remain and is therefore a $O(log(n))$ operation. The justification for the simplifying assumption is that reduction on boolean arrays for a specific dyadic function (in this case, the $\wedge$ operation) can be supported by special hardware (e.g. as in the case of the MasPar MP-1) so that the improved performance can render this a psuedo constant-time operation.

The number of iterations required for `gcd` in the general case depends on the values of the elements in the array to which `gcd` is mapped. The number of iterations required is the maximum of the scalar version of `gcd` applied to each element in the array. This is only true because `gcd` is linearly recursive (and recursive in only one continuation of a conditional statement). In the case of doubly-recursive functions (or linearly-recursive functions which are recursive in more than one continuation of a conditional statement), mapped to an array on a SIMD machine the restriction to a single flow of control becomes costly. SIMD architectures can only support conditional statements by sequentially exploring the different continuations of conditional statements. When these continuations call other functions the entire tree of branching control flow has to be traversed. This can be expensive. This is the price that SIMD architectures pay for a single flow of control.

This graphical method of depicting the efficiency of a particular program is quite useful in developing an intuitive notion of the source of program inefficiency. Consider for example the IFS program in Chapter 6. Because of the number of active locations depends on the type of image that is being drawn, it is difficult to appreciate how efficient/parallel this program is. The efficiency of IFS for the particular image of a fern in Chapter 3 is shown in fig. 6.8. Note that the vertical axis is not continuous (if it were, several pages would be required to display it). Surprisingly, perhaps, this program does *not* make efficient use of PEs on a SIMD machine.

By examining the way in which the IFS algorithm works, the source of the inefficiency becomes clear. Let p be the total number of elements in the plane. Let r be the ratio of elements which are in the image relative to the total number of elements. Let l be

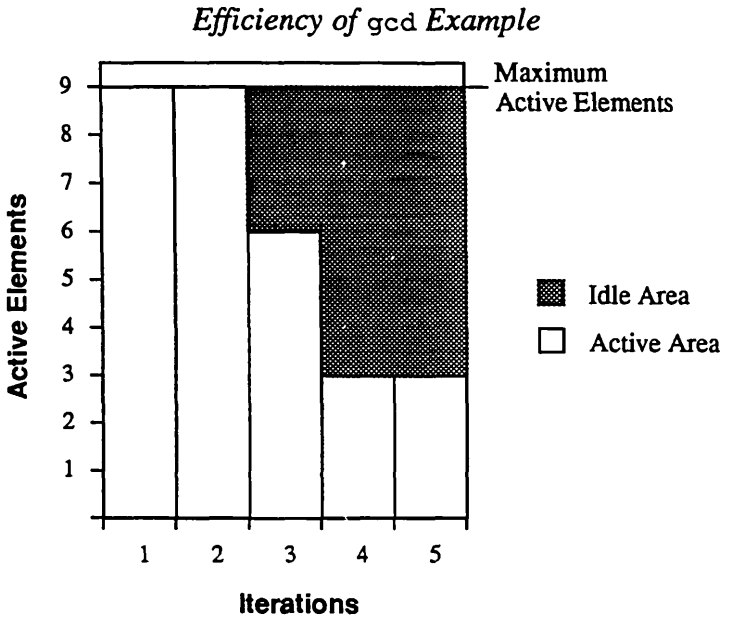


fig. 6.7

the length of the longest unique path through the graph defined by the image.[1] Assume that network contention, caused by multiple messages arriving at a single destination, is non-existent (i.e. `fetch` and `send` take constant time). The average parallelism A, of a breadth-first traversal of such a graph is bounded by:

$$A = \frac{r \cdot p}{l}$$

[1]The longest unique path is the longest path connecting any two points in the graph where each node on the path cannot be visited from another node in the graph at an earlier time by a breadth-first traversal of the graph.

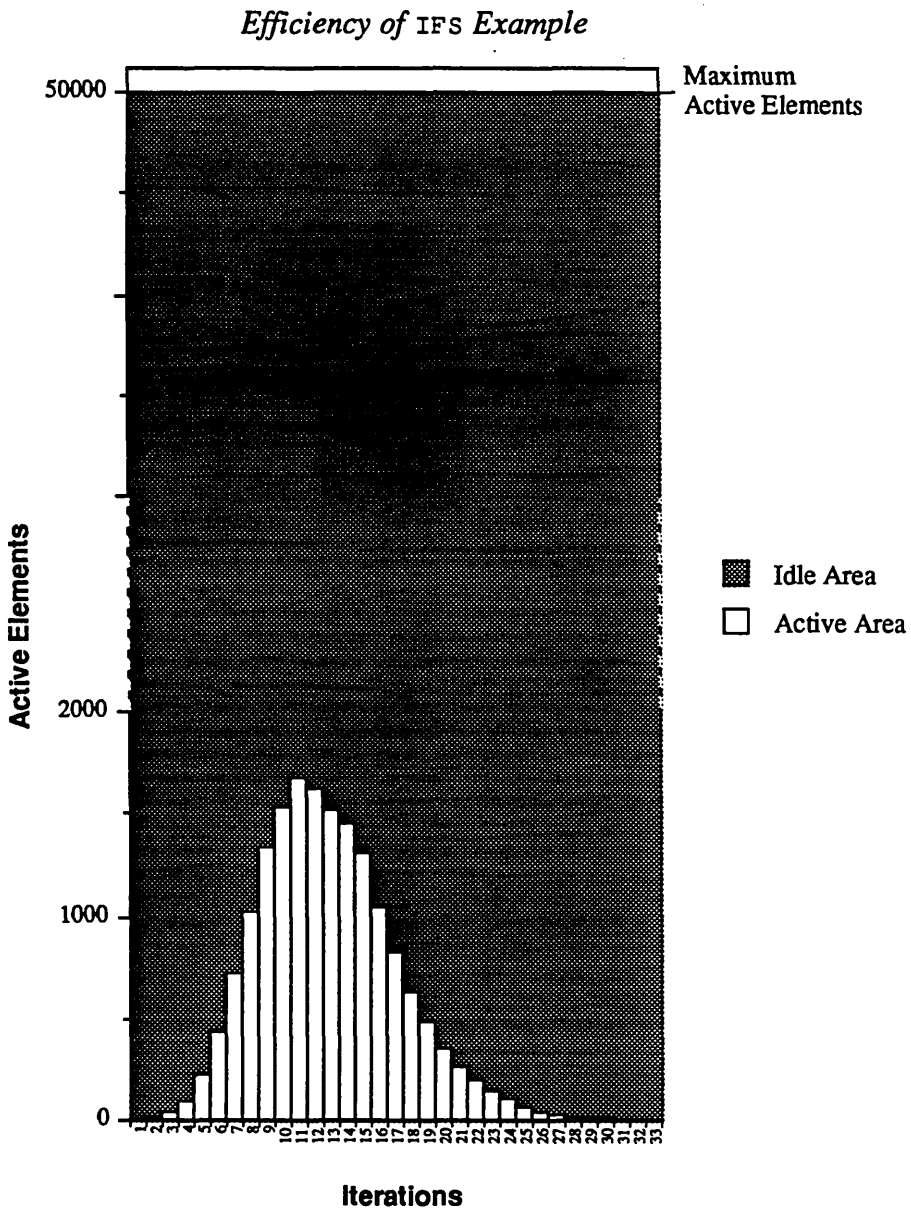


fig. 6.8

This assumes that there is only one pixel per PE. For the fern image in Chapter 3, 17,200 pixels are in the image, out of a possible 50,625. Therefore r = 17,200 / 50,625 and p = 50,625. The number of iterations is equivalent to the longest unique path (l) which is 33. The average parallelism realized by the IFS algorithm in drawing the fern is:

$$A = \frac{17{,}200}{33} \approx 521$$

In this example it is clear the average degree of parallelism is not very high relative to the number of PEs employed. Allocating multiple pixels to PEs would not alleviate the problem since network contention would then dominate. More messages will arrive at each location and communication links would have to be allocated sequentially to resolve

contention. A combining network (see Chapter 2) would not resolve the contention because multiple message for a single PE are addressed to distinct pixels, therefore all messages to distinct pixels need to arrive at their destination.

Tests with other fractal images show that the efficiency of IFS on the fern image is typical. The number of iterations cannot be reduced because it is necessary to traverse the longest unique path through the graph. Allocating multiple pixels per PE does not improve efficiency since contention for communication links will then dominate. The problem lies in the algorithm itself. The long sequence of dependencies (in the IFS example the unique longest path) places a limit on the efficiency of graph-traversal algorithms.

6.3. Implications

The data-parallel operations and language proposed for data-parallel programming were derived following a top-down design process which included the development of an abstract machine architecture (PAM) to execute compiled programs. The code generation examples in this chapter illustrate the consequences with regards to efficiency which stem from those design decisions. The choice of a normal-order reduction strategy was made for the expressive power it provides but it imposes a cost overhead because closures and suspensions must be built. There is a surprising advantage in this reduction strategy in the context of SIMD architectures however which is derived from the way lazy evaluation conserves space.

6.3.1. Space & Time Efficiency

Arrays of algebraic data-types are storage inefficient because in the case of lists, for example, the number of planes that are allocated is equal to the length of the longest list element in the array. This is a consequence of performing resource allocation for all PEs centrally (i.e. by the CPU). This is necessary for synchronous operation. The implications of this can be generalized to arbitrary ADTs.

When planar data-structures are shared, memory occupied by the data-structure cannot be reclaimed until all references to the objects are gone. Shared references to arrays can cause objects in planar memory to persist for long periods of time. When only a single reference to an array exists, it is possible to perform optimizations in the usage of PE memory. This is addressed in §7.1.2.

The presence of conditional statements imposes overheads due to the need to perform reduction on the activity mask to determine whether execution of a `PLANAR_WHEN` block should proceed. There are cases where such tests are clearly unnecessary (i.e. in the absence of any function calls in a continuation). The presence of conditional statements introduces the possibility of branches in control flow which must be traversed sequentially.

The branching of control flow can be reduced in particular instances provided certain conditions are met. This particual issue is discussed in §7.1.3.

6.3.2. Lazy Evaluation

The compilation scheme produces code which performs normal-order reduction. PAM incurs overheads in supporting this reduction strategy. These overheads include double indirections for suspensions (i.e. it is necessary to return to the body of a suspension before a result can be returned to the original caller) and the cost of storing and restoring the context (activity mask, etc.) upon entering a suspension. These overheads stem from the fact that the `PLANAR_OP` instruction is affected by the state of the activity mask. The context needs to be bundled up along with all other free variables in each suspension because the code in the suspension must be executed in the context in which it was created. The context-stack provides storage for the existing context when a suspension is entered. The addition of such a stack and instructions to move the context to and from this stack complicates the abstract machine but even by moving to an applicative reduction order such facilities are required because for conditional statements it is necessary to save and restore the context also. This issue is addressed in §7.1.1.

Aside from expressiveness, there is a particular advantage to a normal-order evaluation strategy and this is that it results in a reduction of the number of "live" objects in memory. Note in the Newton-Raphson example in §6.1.3 how normal-order evaluation avoids calling `newton` to produce the entire list elements in the array. Instead, each element of the lists is demanded, step-by-step as required. During each time step, only one element in each list in the array is "live." Previously computed elements can be deallocated if no longer required. Since the amount of PE memory in present-day SIMD machines is quite limited, this is an important advantage. It is necessary to consider the relative merits of exectution efficiency of applicative evaluation against the space-efficiency provided by normal-order evaluation. Given the advantage of a normal-order reduction strategy in conserving memory, it may be preferable to retain lazyness but remove all redundant suspensions than to forsake normal-order reduction to remove the overhead of suspensions at the cost of increased memory requirements.

Chapter 7

7. Optimization

After compilation, optimization of the produced code is necessary to eliminate overheads and increase performance. Programs can be optimized in two ways: automatically by the compiler, or with the assistance of the programmer (e.g. through the use of program transformation methods). Functional programs admit more scope for optimization because of their mathematically sound semantics. In the previous chapter the efficiency aspects of specific language features and algorithms were considered. Efficiency was seen to comprise: inherent machine characteristics, the compilation system, and algorithm choice. This chapter focuses on the last two aspects and categorizes a number of optimizations which may be carried out to boost performance.

The automatic optimizations applicable to compiling programs for PAM consist of adaptations of existing techniques such as strictness and sharing analysis and also several novel optimizations unique to a synchronous data-parallel architecture such as PAM including the elimination of unnecessary bifurcation of control flow and a way of optimizing loop termination.

The data-parallel operations provided in Chapter 2 allow for the development of a set of axioms for transforming compositions of data-parallel operations. An initial algebra for the manipulation of data-parallel programs is developed which includes an adaptation and extension of the Parallel Data Transform work by Flanders. This permits algorithms using inefficient forms of communication to be transformed into efficient equivalents. This leads to the possibility of creating a new methdology for deriving highly-efficient data-parallel algorithms.

7.1. Automatic Optimizations

Automatic optimizations usually consist of a compiler being able to identify and replace inefficient code with efficient equivalents. More substantail modifications of the original source program rely on insight and application-specific information which requires the programmer to guide the optimization effort towards the more efficient target form of the program.

In imperative languages, automatic optimizations usually take the form of optimizations at the object-code level. The clean and simple semantics of functional languages enables the compiler to use more high-level optimizations at the source-language level as well because it can be shown that these will never change the meaning of a program. Functional languages, in particular those using a normal-order reduction strategy, present a variety of challenges to the production of efficient code. The planar instruction mode of PAM introduces scope for a novel set of automatic optimizations, including the elimination of unnecessary bifurcation of control flow, reducing the overhead incurred by conditional statements, and a new way of optimizing loop termination.

7.1.1. Strictness Analysis

Lazyness in a functional language allows the use of infinite data structures and the implementation of interactive I/O while doing the minimum amount of evaluation necessary to yield the desired result. The effect of lazy evaluation in a machine such as PAM (which exploits data-parallelism) is less detrimental to performance than in MIMD graph-reduction systems (which exploit process-parallelism). Lazyness discourages concurrent activation of processes because of the demand-driven nature of normal-order reduction. Parallelism in PAM comes from the execution of monolithic operations, not from the concurrent activity of independent processes. Lazyness does not affect the degree of parallelism exploited in planar PAM instructions.

In graph-reduction systems, lazyness reduces opportunities for parallelism so strictness analysis has been proposed as the philosopher's stone which will turn base (lazy) programs into gold (parallel & efficient). Strictness analysis is required in these systems to gain parallelism *and* to reduce the overhead of allocating closures (see [Clack85] for a discussion on strictness analysis and [Burn90] for use of strictness analysis in a parallel graph-reduction architecture).

A function f is said to be strict if $f \perp \equiv \perp$. In other words, if the evaluation of an argument does not terminate ($\perp$) then a function is strict if when supplied with such an argument it also does not terminate. Knowing that a function is strict means that arguments can be reduced *eagerly*, using an applicative-order reduction scheme (i.e. the strict arguments are reduced before the function is called). It is therefore unnecessary to

construct closures for expressions passed as arguments of functions known to be strict. In an example in the previous chapter, `gcd` was defined as:

```
gcd = λu.λv.
   case (= v 0) in
      when true : u
      when false : gcd v (mod u v)
```

it is clear that `gcd ⊥ v` ≡ ⊥ and `gcd u ⊥` ≡ ⊥ for any arguments `u` and `v` (i.e. `gcd` is strict in both of its arguments). In the PAM code generated for `gcd`, a suspension is built for the expression `(mod u v)` and a pointer to this suspended expression is then passed to the recursive call to `gcd`. Strictness analysis indicates that the construction of such a suspension is clearly unnecessary and it is possible to evaluate `(mod u v)` *before* calling `gcd` without changing the semantics of the function.

In the presence of conditional statements, conventional strictness analysis schemes assume that an argument is strict only if it is evaluated in all continuations of a conditional statement. If it is absent from even one of the continuations it must be assumed that the argument may not be evaluated eagerly. In the planar code stream of PAM more than one continuation may be executed. Any arguments in continuations where the activity mask contains active elements will be evaluated. Consequently, more opportunities for eliminating redundant suspended evaluations exist in the planar code stream than in the scalar one.

For example, in the function `f`:

```
f = λx.λy. (= x 0) → x; (+ y x)
```

the scalar rendition of `f` is only strict in `x` whereas the planar version of `f` is strict in both `x` and `y`, *provided* that it can be determined that some elements in the plane `x` are non-zero. Strictness analysis for the planar renditions of functions can uncover more strictness when the analysis is able to predict the presence of active elements in the activity mask of a continuation. In the planar mode there are fewer "places" where lazyness can lurk. In the presence of algebraic data-types, it is often the case that one or more of a constructor's arguments are discarded. For example, the `length` function, which returns the length of a list:

```
length = λxs. case xs in
   when nil : 0
   when cons h t : (+ 1 (length t))
```

This function doesn't need to evaluate the elements in the list, it only needs to know how many there are! Discovering which arguments of a constructor will always be required requires a special form of strictness analysis. One example of this is the *Evaluation Transformer Model* [Burn87]. In the presence of higher-order functions it is

impossible to decide (at compile-time) which functions will be applied to an argument. Constructing suspensions therefore remains necessary.

As stated in the previous chapter, there is an auxiliary benefit to using lazy evaluation for PAM in that it helps to limit the amount of planar memory required. This is an important consideration in any implementation of PAM on a real SIMD architecture as the amount of memory per PE in these systems is often quite limited. The overhead of allocating suspensions may be acceptable because normal-order reduction conserves scarce PE memory.

7.1.2. Reference Counts/Sharing Analysis

The demand-driven nature of normal-order reduction means that the producer and consumer of constructed data-types operate in synchrony: a demand by the consumer leads to the producer producing another element of the data structure. Applicative-order evaluation leads to the completion of the producer before the consumer function is evaluated. Where data-structures which are single-threaded (i.e. no shared references exist), the space occupied by each produced element can be discarded as soon as the consumer no longer requires it. The "write-once" nature of functional languages ensures that the lifetimes of data-objects are often very short. This leads to a rapid turn-over of objects in memory. If a data-object has only a single reference, then it is safe to re-use the memory allocated to it to perform *destructive updates* without loss of referential transparency.

The synchronous nature of PAM's planar instruction stream ensures that planes which are single-threaded can be updated destructively because all locations are written concurrently. Because communication and computation are strictly separated and elements in adjacent locations are inaccessible (due to the strict enforcement of locality), there is no possibility of an inconsistent machine state occurring by destructively updating single-threaded arrays. In data-parallel models where concurrent computation and communication are *asynchronous*, it is necessary to ensure that all locations read contain the "correct" value (i.e. locations should only be read after they have been written). In asynchronous models it is necessary to perform some form of explicit synchronization to ensure consistency.

PAM is a synchronous machine so destructive updates on single-threaded arrays require no additional synchronization. The number of references to a data-object can be discovered at compile-time by some form of analysis (e.g. *Path Analysis* [Bloss89] or compile-time reference-counting [Hudak86]), or by run-time reference-counting schemes. Counting references at run-time entails updating a counter associated with each data-object every time a reference is copied/discarded [Glaser88]. For scalar objects this is an unacceptable overhead. In PAM's planar mode, it is only necessary to associate a single

reference count with an entire plane (see Chapter 4) because references can only refer to planes and not to individual elements within a plane (i.e. all elements in a plane have the same reference count). The cost of maintaining reference counts is amortized against the benefit of data-parallel computation on the planar object. In other words, updating references to planar variables incurs an overhead for each planar operation but such operations are massively parallel, equivalent to perhaps thousands or tens of thousands of individual scalar instructions. Lazy-evaluation in conjunction with some form of analysis or mechanism to discover when planar objects are shared is worthwhile in PAM because this keeps the amount of planar memory required to a minimum.

7.1.3. Eliminating Bifurcating Control-Flow

If different continuations of a conditional statement contain function calls then control is said to *bifurcate*. PAM explores the bifurcation by traversing each branch of the flow of control in turn. Branches of control flow are serialized in PAM, reducing data-parallelism. There are cases where such bifurcation is clearly unnecessary.

For example, if different continuations of a conditional statement call the *same* function, it is possible to remove the recursive call from the continuations to eliminate the branch of control. Consider the function `d2b`:

```
d2b = λn. case n in
   when 0 : nil
   otherwise : case (mod n 2) in
                  when 0 : cons 0 (d2b (/ n 2))
                  when 1 : cons 1 (d2b (/ (- n 1) 2))
```

This function converts the decimal representation of a number into its base 2 representation as a list of binary digits. Both branches of the second conditional contain a call to `d2b`. This version of the function requires PAM to bifurcate control flow during each iteration as each branch contains a separate call to `d2b`. Each continuation requires a separate test of the activity mask to see if any active locations remain. However, it is possible for a compiler to transform this version of `d2b` into the equivalent form below:

```
d2b = λn. case n in
   when 0 : nil
   otherwise : let (d, a) = case (mod n 2) in
                               when 0 : (0, (/ n 2))
                               when 1 : (1, (/ (- n 1) 2)) in
                  cons d (d2b a)
```

The modification does not change the scalar semantics of the function. However, in the planar rendition of `d2b`, `d` and `a` are planar objects and the conditional statement will set different elements in `d` and `a` to their appropriate values as arguments to `cons` and `d2b`. Only a single call is now required and control flow does not bifurcate, which leads to fewer planar instructions being executed in order to reach termination. Furthermore, the second

`case`-statement can now be optimized to remove the test of the activity mask in each of its continuations, as described in the next section. The optimization performed on `d2b` is applicable wherever the compiler detects a conditional statement of the form:

```
case e0 in
        when c1 ... : e1[f x11 ... x1n]
        when c2 ... : e2[f x21 ... x2n]
            .
        when cm ... : em[f xm1 ... x2n]
```

Where the notation $e[f\, x_1 \ldots x_n]$ represents an expression e containing a sub-expression consisting of a call to the function f with arguments $x_1 \ldots x_n$. Provided that each continuation of a conditional statement contains a call to f, then it may be possible to compute the arguments for the function call separately and call f only once. The optimization consists of translating a conditional statement of the above form into:

```
let (y1, ..., yn) = case e0 in
                        when c1 ... : (x11, ..., x1n)
                        when c2 ... : (x21, ..., x2n)
                            .
                        when cm ... : (xm1, ..., x2n) in e[f y1 ... yn]
```

This optimization can only be performed if the enclosing expression in each continuation is the same. This expression (e) is modified to contain a call to f which takes the computed planar values $y_1 \ldots y_n$ as arguments. In the case of `d2b`, the enclosing expressions in the continuations are:

```
e1 = cons 0 (d2b (/ n 2))
e2 = cons 1 (d2b (/ (- n 1) 2))
```

The enclosing expressions are not strictly equivalent because the first argument to `cons` in each expression is different. However, this differing parameter can also be "lifted-out" and returned by the `case`-statement enclosed by `let`. The restriction on the equivalence of $e_1 \ldots e_n$ can be relaxed to requiring that only the outermost expression of $e_1 \ldots e_n$ must be the same. Any differing sub-expressions can be placed in the conditional statement and returned by the `let`. The most common occasion where this type of optimization can be used is in the case of tail-recursive functions where the recursive call is the top-level expression. This optimization can then be performed automatically by a compiler to ensure that there is no unnecessary serialization due to the bifurcation of control flow.

7.1.4. Optimizing Loops

Where the continuation of a conditional statement is known not to involve recursion, it is unnecessary to carry out the reduction on the activity mask that PLANAR_WHEN ordinarily performs. If all of the elements in the activity mask of a continuation are set to false, then none of the PEs can respond to planar instructions so such a continuation has no effect. A compiler can readily detect cases where continuations are non-recursive and replace the conventional form of PLANAR_WHEN with one that does not perform the test on the activity mask. The most common example is in the case where a continuation consists entirely of basic arithmetic/logical expressions. A compiler can discover which functions do not call any other functions from the construction of a program *call-graph.* A call-graph permits the detection of cycles in a program. All functions which do not form part of a cycle can use the optimized form of PLANAR_WHEN in the compilation of conditional statements.

Even in cyclic functions, it may be possible to reduce the required overhead. For example, consider a program to calculate prime numbers using a data-parallel implementation of the sieve of Eratosthenes:

```
sieve1 = λar.
   let br = imap (λj.λt. (2, j, t)) ar in
       map (inc ^ (lim ar)) br

inc = λ(i, j, t). ((+ i 1), j, (test i j) → false; t)

test = λi.λj.
   (∧ (> j i) (= (mod j i) 0))

lim = λar.
   ceil (sqrt (bound ar))
```

The array `ar` is an array of booleans. Initially, all elements are set to true. An array `br`, is derived from `ar` by pairing each boolean value (`t`) with its index (`j`), and the constant `2`. When `sieve1` returns, the prime numbers are all those elements where the boolean field in the tuple remains true. The program maps an iterated function (`inc`) over the elements in `ar`, which eliminates all of the elements where the iteration number, `i`, is a divisor of `j` (this is implemented by use of the `mod` function). To prevent prime numbers from being eliminated (since a prime number has itself as a divisor), a number is only eliminated if the iteration number (`i`) is less than the number (`j`). The number of iterations of `inc` applied at every element is the same (`lim ar`).

In this version of the algorithm, no communication is required. The `sieve1` eliminates all elements with divisors up to the square root of the maximum index in the array.[1] The presence of recursion in the function being mapped: `(inc ^ (lim ar))`

[1]This algorithm has parallel complexity asymptotic to $O(\sqrt{n})$. Alternative implementations with lower complexity exist, but they require communication. This version of the sieve can be expressed

prevents the elimination of the reduction test on the activity mask. This function belongs to a particular class of recursive functions where the number of iterations at each element is the same for all elements in the array. Hence, it is possible to "lift" the recursion out of the mapped function and iteratively apply a non-recursive function to compute the same result, e.g. as in `sieve2`:

```
sieve2 = λar.
   (elim ^ (lim ar)) (2, ar)

elim = λ(i, ar).
   ((+ i 1), imap (λj.λt. (test i j) → false; t) ar)
```

In this variation of the program, the function `elim` is applied `(lim ar)` times. The `elim` function is not recursive (so it is unnecessary to reduce the activity mask prior to executing each continuation). The cycle in the planar code stream in `sieve1` has been changed into a cycle in the scalar code stream in `sieve2`. Termination detection in scalar mode involves testing a scalar value which is a constant-time operation.

It is probably beyond the capabilities of present compilers to perform the sophisticated manipulations required to derive `sieve2` from `sieve1`. Nevertheless, this derivation relies only on sophisticated program-manipulations and not on insight. Therefore, the potential exists that future compilers may detect and optimize functions of such as `sieve1` without assistance from the programmer.

It is possible to improve the program still further. The presence of a conditional statement in `elim` continues to present overheads because arguments on the stack are copied and partial planar results are returned by each continuation which must be combined at the end of each iteration. It is worthwhile to consider whether it is possible to remove the conditional statement altogether.

The final version of the sieve program, `sieve3`, maps a non-recursive, unconditional function over the array:

```
sieve3 = λar.
   (comp ^ (lim ar)) (2, ar)

comp = λ(i, ar).
   ((+ i 1), imap (λj.λt. (∧ (¬ (test i j)) t)) ar)
```

`comp` is derived from `elim` by making use of the following identity:

$$(p \rightarrow q;\ r) \equiv (p \wedge q \vee \neg p \wedge r)$$

A compiler can make use of the information provided by the type-checker to detect conditional statements returning boolean results and can transform them into equivalent expressions using boolean operations. This is a simple optimization which can be

---

more concisely by mapping a recursive function into the array. It is possible to derive the version given here from the recursive formulation by the use of transformation techniques.

incorporated into a compiler without introducing substantial additional complexity. The body of the lambda-expression mapped onto `ar` in `comp` generates the following PAM code:

```
w0 := EVAL_PLANAR { ENTER_PLANAR test [j, i] }
w1 := PLANAR_OP (¬) [w0]
w2 := EVAL_PLANAR { ENTER_PLANAR t [] }
w3 := PLANAR_OP (∧) [w1, w2]
RETURN_PLANAR w3
```

Which involves no conditional code (i.e. `PLANAR_WHEN`) whatsoever.[2] The planar code for `test` is also free of conditional code:

```
test := COMBINATOR ({
    .
    .
    .
}, {
    POP j
    POP i
    w0 := EVAL_PLANAR { ENTER_PLANAR j [] }
    w1 := EVAL_PLANAR { ENTER_PLANAR i [] }
    w2 := PLANAR_OP (>) [w0, w1]
    w3 := EVAL_PLANAR { ENTER_PLANAR j [] }
    w4 := EVAL_PLANAR { ENTER_PLANAR i [] }
    w5 := PLANAR_OP (mod) [w3, w4]
    w6 := MAKE_PLANAR 0
    w7 := PLANAR_OP (=) [w5, w6]
    w8 := PLANAR_OP (∧) [w2, w7]
    RETURN_PLANAR w8
})
```

The resulting code benefits from being freed of all the overheads associated with conditional statements and planar reduction tests to detect the termination of recursion. The optimizations presented here are quite high-level and significantly more complex than the type of optimizations commonly performed by existing compilers for imperative-languages. The division between object-code and source-code optimizations becomes blurred and optimizations may sometimes require the assistance of the programmer. At this point, optimization becomes a programmer-assisted task and leads into the realm of program transformation.

7.2. Optimizations via Program Transformations

Automatic compiler optimizations can only go so far towards improving programs. Far greater gains are to be had from removing inefficient parts of a program altogether or by

---

[2]The argument to comp is a pair of values and in Chapter 2 tuples were regarded as syntactic shorthand for a built-in algebraic data-type. Since conditional statements are the only way in which ADTs are decomposed, an implementation of tuples as ADTs would needlessly re-introduce a conditional statement (with only one continuation). Conditionals with only a single continuation can be optimized to yield unconditional code.

modifying the algorithm used in a program. These types of optimizations are typically beyond the ability of automatic means (however, some very high-level transformation techniques can be automated to a large extent as *transformation tactics* [Wei-Ng90]). At this level optimizations become the preserve of the programmer, armed with a methodology (or two) for transforming programs and guided by insight, experience, and intuition aided by the transparent nature of the primitive language constructs.

Program transformation is a large area of research and it is beyond the scope of this thesis to attempt to develop a complete methodology for the transformation of data-parallel programs. Instead, some simple axioms and lemmas are presented as an initial set of tools used to improve data-parallel programs. The development of more powerful lemmas and transformation systems applicable to data-parallel programs is a promising area for future research.

7.2.1. Introduction to Program Transformation Methodologies

In recent years two main transformation methodologies have emerged. The *unfold/fold* methodolgy developed by Burstall and Darlington [Bursta77] and the *algebraic* methodology espoused by Backus [Backus78], Bird and Meertens (e.g [Bird87], [Meerte86], [Meerte89]), and Harrison [Field88] among others. These two, different, approaches to program transformation can be combined to develop a methodology suitable for the transformation of data-parallel functional programs.

7.2.1.1. The Unfold/Fold Methodology

The unfold/fold methodology is based on a set of 6 rules stated as program equivalences which permit a style of program transformation akin to performing structural induction. The 6 rules are given below:

- *Definition*: Define a new function.
- *Instantiation*: Instantiate some of a function's inputs.
- *Unfolding*: Replace a function call with the body of the function.
- *Folding*: Replace an expression corresponding to the body of a function with a call to the function.
- *Abstraction*: Lift out a sub-expression into a `let` clause and replace each occurrence of the sub-expression with the variable returned by the `let`.
- *Laws*: Replace any expression by an equivalent form justified by axioms or lemmas.

The unfold/fold style of transformation derives its elegance from the use of "patterns" in the definition of functions. This obviates the need for many common conditional statements to test the values of arguments. For the remainder of this chapter constructors and constants are allowed to occur on the left-hand side of lambda-expressions because this makes transformations using the unfold/fold methodology more succinct. All such patterns can be compiled out into explicit tests on the arguments by `case`-statements [Peyton87b].

7.2.1.2. The Algebraic Approach

The algebraic style of program transformation relies on a set of equivalences of program forms which are stated as axioms or lemmas. There are a number of instances of where such laws can be used to simplify the transformation process. Transformation by the exclusive application of axioms or laws is what is known as the algebraic style of transformation. The algebraic approach can be considered a specific aspect of the more general unfold/fold methodology but the application of axioms or laws can be automated and implemented by sophisticated compilers independently of the general unfold/fold framework.

Some useful functions and axioms are defined below:

```
id = λx. x

f × g = λ(x, y). (f x, g y)

first = λ(x, y). x
second = λ(x, y). y
```

Initial Axioms

P1:	$f \circ (\lambda x.\, e) \equiv \lambda x.\, f\, e$		
P2:	$f \circ (p \rightarrow q; r) \equiv p \rightarrow f\, q; f\, r$		
P3:	$\lambda x.\, \text{let } y = e_1 \text{ in } e_2 \equiv (\lambda y.\, e_2) \circ (\lambda x.\, e_1)$		iff $x \notin \mathcal{FV}[\![e_2]\!]$
P4:	$f \circ first \equiv first \circ (f \times id)$		
P5:	$f \circ second \equiv second \circ (id \times f)$		

7.2.2. An Algebra for Data-Parallel Program Transformation

The data-parallel primitives in Chapter 2 do not have source-language definitions. It is not possible to use the unfold/fold methodology to transform compositions of these operations because it is impossible to decompose (unfold) the operations. Instead, it is necessary to present axioms so that the algebraic style of programming can be used. A set of axioms of the primitive data-parallel operations is given below:

Axioms of Data-Parallel Primitives

P6:	$map\ id \equiv id$
P7:	$map\ f \circ g \equiv (map\ f) \circ (map\ g)$
P8:	$map\ (\lambda x.(e_1, e_2))\ ar \equiv zip\ (map\ (\lambda x.\, e_1)\ ar)\ (map\ (\lambda x.\, e_2)\ ar)$

P9: $(map\ f)\circ(rotate\ i) \equiv (rotate\ i)\circ(map\ f)$
P10: $f\circ(select\ i) \equiv (select\ i)\circ(map\ f)$

Some of these axioms arise naturally from a categorical foundation of aggregate data-structures, as discussed in Chapter 2. Using these axioms it is possible to prove powerful general lemmas. For example, consider the notation for iteration that was adopted in Chapter 2: `f ^ n`. The iteration operation (^) can be defined as an infix function as:

```
f ^ 0 = id
f ^ n = f∘(f ^ (- n 1))
```

Using this definition the correctness of the iteration-promotion optimization proposed in §7.1.4 can be proved:

```
g = λi.λar. map (f ^ i) ar

g = λ0.λar. id                                          inst.+unfold (^) i=0
  | λn.λar. map (f∘(f ^ (- n 1)) ar                     inst.+unfold (^) i=n
  ⇒         (map f)∘(map (f ^ (- n 1))) ar                        P2
  ⇒          map f (map (f ^ (- n 1)) ar)                    unfold ∘
  ⇒          map f (g (- n 1) ar)                        fold with g
  ⇒         (map f)∘(g (- n 1)) ar                            fold ∘

g = λ0.λar. id
  | λn.λar. (map f)∘(g (- n 1)) ar                      fold with (^)
  = (map f) ^ n
```

This transformation is proof of the lemma:

L1: $map\ (f\ \hat{}\ i) \equiv (map\ f)\ \hat{}\ i$ (*iteration promotion lemma*)

In the original expression, `map (f ^ n)`, each PE operates on its own (local) copy of the value `n`. Each PE then individually decrements this value and each recursive call to `f` involves performing a reduction on the activity mask to see whether any active PEs remain. This is clearly wasteful since all PEs will become inactive at exactly the same iteration. In the transformed equivalent `(map f) ^ n`, the termination condition relies on testing the scalar value `n`. The first version requires performing an $O(lg(n))$ reduction for each call to `f` whereas the second involves a simple scalar test with O(1) complexity.

7.2.3. Transparency Aids Transformation

Transparency aids the derivation of more efficient programs through transformation. The primitive operations have a uniform operational semantics and the cost associated with their use is known. It is possible to apply the axioms introduced previously to derive

more efficient programs by eliminating costly components of a program with equivalent components of reduced complexity. The transparent nature of the primitive operations allows the programmer to compare alternative formulations of an algorithm with regards to efficiency and parallelism.

Consider the histogram example from Chapter 3. An initial specification of this problem might be as follows:

```
hist1 = λar.
   map length
       (imap (λj.λxs. filter (λx. (= (ord x) j)) xs) (copy ar))
```

where `copy ar` produces an array of lists with all the elements of `ar` at each location. This can be implemented as:

```
copy = λar.
   let (_, ars) = (dup ^ (+ 1 (bound ar)))
                       (ar, map (λa. nil) ar) in ars

dup = λ(ar, ars). (rotate 1 ar, map2 cons ar ars)
```

Which rotates the array `ar` n times (where n is the number of elements in `ar`) and accumulates the elements together into a list. The `length` and `filter` functions are defined as:

```
length = λxs. fold (λx.λn. (+ n 1)) 0 xs
filter = λp.λxs. fold (λx.λys. p x → (cons x ys); ys) nil xs
```

This specification of the histogram problem states that by placing a list of all the elements of the array `ar` at each location, filtering out those where the ASCII value of the character is equivalent to the index, and counting the number of elements that remain produces a histogram of the characters in the original array. This specification is inefficient because `copy` must send each character to every location in the array. By use of transformation, it is possible to yield a more efficient version of this algorithm.

Starting with `hist1`:

```
hist1 = λar. imap (λj.xs. length (filter (λx. (= (ord x) j) xs)))
                  (copy ar)                                    P1 & P7
      ⇒       imap (λj.xs. (fold (λy.λn. (+ n 1)) 0)
                           (filter (λx. (= (ord x) j) xs)))
                  (copy ar)                                    unfold length
      ⇒       imap (λj.xs. (fold (+)∘(λz. 1) 0)
                           (filter (λx. (= (ord x) j) xs)))
                  (copy ar)                                    identity
```

From the definitions of `fold` and `maplist` it follows that:

```
fold (+)∘(λz. 1) 0 ≡ (fold (+) 0)∘(maplist (λz. 1))
```

Defining sum as follows:

```
sum = λar. fold (+) 0 ar
```

permits the transformation to continue:

```
⇒       imap (λj.xs. sum∘(maplist (λz. 1))
                     (filter (λx. (= (ord x) j) xs)))
             (copy ar)                                  defn. & fold with sum
⇒       imap (λj.xs. sum∘(maplist (λz. 1))
                     (fold (λx.λys. (= (ord x) j) →
                           (cons x ys); ys) nil xs)))
             (copy ar)                                  unfold filter
⇒       imap (λj.xs. sum∘(maplist (λz. 1))∘
                     (fold (λx.λys. (= x j) →
                           (cons x ys); ys)∘(ord) nil) xs)
             (copy ar)                                  defn.
⇒       imap (λj.xs. sum∘(maplist (λz. 1))∘
                     (filter (λx. (= x j)))∘
                     (maplist (ord)) xs)
             (copy ar)                                  defn. & fold with filter
⇒       imap sum∘(maplist (λz. 1))∘
              (λj.λxs. (filter (λx. (= x j)))∘
                       (maplist (ord)) xs)
             (copy ar)                                  defn. & fold with filter
```

Note how immediately after replicating all the elements in the array, `(maplist (ord))` converts all the elements to their ASCII values. This conversion can be promoted into the copy operation by assuming the following identity:

```
(map (maplist f)) (copy ar) ≡ copy (map f ar)
```

Hence:

```
⇒       imap sum∘(maplist (λz. 1))∘
              (λj.λxs. filter (λx. (= x j)) xs)
             (copy (map ord ar))                        identity
```

To proceed further it is necessary to remove the ineffiency of copy. Every character has a single ASCII value. Filtering all of the elements whose ASCII value corresponds to the index of the current location is equivalent to sending every element to the location corresponding its ASCII value directly. This resembles the definition of send:

$$\texttt{send} \ll i_0, \ldots, i_n \gg \ll x_0, \ldots, x_n \gg \Rightarrow \ll [x_r \mid i_r{=}0], \ldots, [x_s \mid i_s{=}n] \gg$$

By assuming the equivalence of copy with filter and a send where the destinations of elements correspond to their ASCII value, it is possible to continue:

```
⇒       imap sum∘(maplist (λz. 1))
             (send (map ord ar) (map ord ar))           defn.
```

There is a useful property of `send`, which is derived from its definition:

```
send dr (map f ar) ≡ map (maplist f) (send dr ar)
```

This indicates that elements in `ar` can be converted to their ASCII value *after* they have been sent to their destinations:

```
⇒   imap sum∘(maplist (λz. 1))∘(maplist ord)
        (send (map ord ar) ar)                          defn.
⇒   imap sum∘(maplist (λz. 1))
        (send (map ord ar) ar)                          defn.
⇒   imap sum
        (send (map ord ar) (map (λz. 1) ar))            defn.
⇒   imap (fold (+) 0)
        (send (map ord ar) (map (λz. 1) ar))            unfold sum

⇒   scatter (+) 0 (map ord ar)
          (map (λz. 1) ar)                              fold with scatter
```

The result is that by using the unfold/fold methodology, plus some axioms and properties of the functions involved, the histogram algorithm provided in Chapter 2 has been derived from the original specification. The `copy` operation has complexity O(n) and `filter` and `sum` also have complexity O(n). The `map` operation has parallel complexity equivalent to the function being mapped, therefore the inefficient formulation of the histogram problem, `hist1`, has complexity O(n).

The time required to execute the efficient version is determined by `send`, which depends upon the number of elements routed to a single location and the presence/absence of a combining network (see Chapter 2). On a machine without a combining network, the time required is proportional to the number of times the most frequent character occurs in the original array. With a combining network, where elements can be summed along the way, the time required is proportional to the logarithm of the number of times the most frequent character occurs. The transformed form of `hist1` is more efficient and contains greater opportunity for the exploitation of parallelism.

7.2.4. Eliminating Inefficient Communication

Transformations on data-parallel functional programs can be divided into two classes: those that attempt to minimize the cost of computation and those that minimize the cost of communication. The primitives in Chapter 2 provide two forms of communication, characterized by `rotate` on the one hand and `send`/`fetch` on the other. The second form of communication is more general but is more costly than the first, requiring more sophisticated interconnection capabilities which are typically not contention-free.

A promising application of program transformation entails expressing certain patterns of communication described via `fetch` and `send` as a series of `rotate` or `shift` operations. An initial foray into this area is presented in [Jouret91b].

This work is motivated by the existence of regular patterns of communication in many numerical algorithms and the ease with which some regular communication patterns can be implemented on locally-interconnected processor-arrays (e.g. [Jessho80]). Flanders has developed a simple methodology called *Parallel Data Transforms* (PDT) which can automatically generate the `rotate/shift` operations required to implement more general, global, communication patterns [Flande87]. The communication patterns to which this methodology is applicable regards those where the pattern can be specified as a boolean transformation on the source address of an element. The kinds of communication patterns that can be described in this way include perfect shuffles, sheet/crinkle-mappings, exchange networks, etc.

Flanders examined the class of routings that could be derived from bit-wise

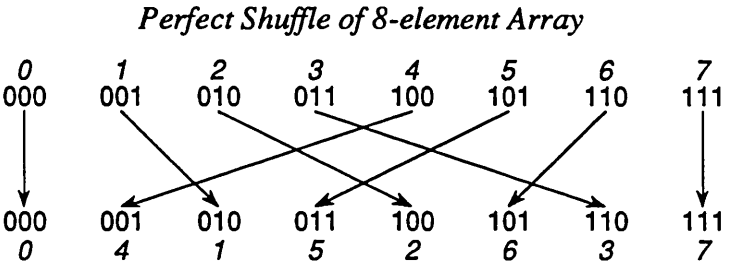


fig. 7.1

manipulations of the source index of an element to yield the destination index. Such routings rely on manipulating the index of a location and are *value-invariant* (i.e. insensitive to the value of the element being routed). Such routings can be defined using an abstraction of the `send` primitive, `route`:

```
route = λf.λar. send (imap (λj.λa. f j) ar) ar
```

The function `f` computes a destination index for each source index. In his PDT paper, Flanders restricts the type of `f` to functions consisting of compositions of the following two operators which operate on the binary representation of the source index:

- $(e\ p\ q)$: exchanges the bits at positions p and q.

- $(i\ p)$: inverts the value of the bit at position p.

For example, the perfect shuffle on an array of 8 elements consists of reordering the bits of a source index, $b_2b_1b_0$ to compute its destination index, $b_1b_0b_2$. The perfect shuffle can be implemented using `route` (and is shown in fig. 7.1:

```
route ((e 0 1)∘(e 1 2)) ar
```

First bits 1 and 2 are exchanged and then bits 0 and 1. The existing bit-operations can be used to generate $n \cdot lg(n)!$ different routings (where n is the number of elements being routed and which is a power of 2). The appeal of this particular method is based on the realization that a routing defined by using these simple bit-manipulation operations can

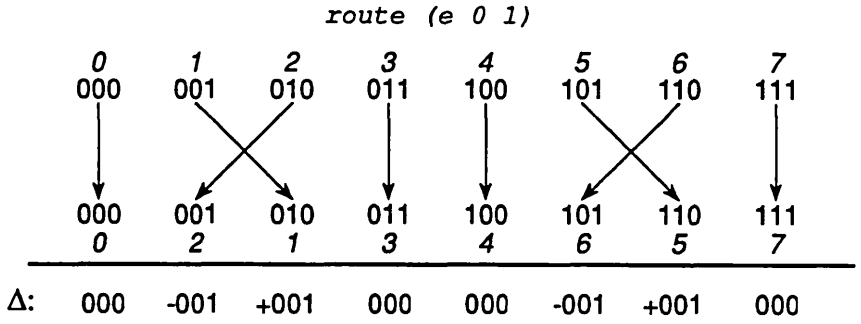


fig. 7.2

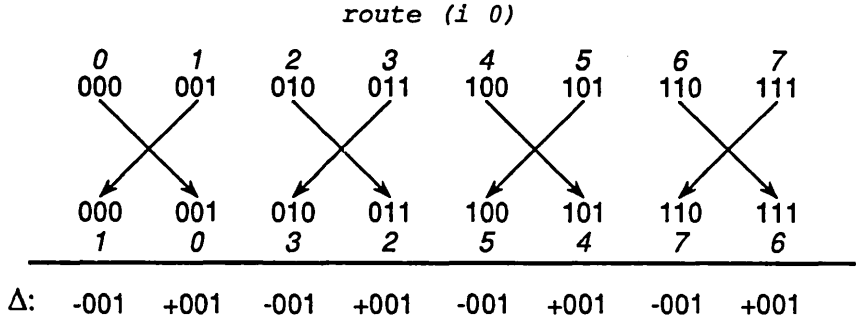


fig. 7.3

be efficiently realized by compositions of `rotate`.

For example, consider what happens when `route (e 0 1)` is applied to an array of 8 elements. The result is shown in fig. 7.2. The binary representation of each index is shown underneath the index. For each location, the displacement from the source location to the destination location is shown in the row labelled Δ. A displacement of 000 corresponds to no data-movement, -001 is equivalent to `rotate -1` whereas +001 corresponds to `rotate 1`. For the example in fig. 7.2 it can be seen that this particular data-routing can be implemented using two `rotate` operations followed by an operation to merge the elements from the resulting arrays into one. A similar example is shown for the

case of bit-inversion in fig. 7.3. This is even simpler in that the merge phase only takes elements from the rotated arrays (and not from the original, un-rotated, array).

For each of the simple bit-manipulation expressions used as a routing function it is possible to define an equivalent function using `rotate`. These functions are denoted by the identifiers `E` and `I` which correspond to the bit-operations *e* and *i* (used in `route`) respectively.

What is lacking is an operation which allows elements to be routed to a common destination. This requires an extension of the PDT system. A new bit-operation is introduced which allows an individual bit-position to be set to either 0 or 1. This operation is (*s p v*) which sets the bit at position *p* to the value *v*. For example, the operation:

```
route (s 0 0) ar
```

has the effect of pairing up all the adjacent array elements. So for the array used in fig. 7.1 where each array element contains the value of its index, the result will be an array with the elements: [0,1], [], [2,3], [], [4,5], [], [6,7], []. As with the previous bit-manipulations, there is an equivalent efficient form, denoted by the upper-case symbol `S`:

```
S (<>) (λx. nil) p v
```

This operation is distinct from the previous ones in that it is parameterized by two functions, one to combine elements arriving at the same location (in this case, the append function on lists: <>), and the other which is applied to elements where no other elements arrive `(λx. nil)`. By making these two functions parameters of `S`, it is possible to promote computation into data-movement as will be shown later.

The benefit of breaking down the routing function into compositions of these simple bit-operations is that compositions of the bit-operations in `route` corresponds to compositions of the `E`, `I`, and `S` functions defined using fast `rotate` operations. A translation can be performed which converts composition of these expressions in `route` to compositions of efficient `E`, `I`, and `S` functions. These functions are defined below:

```
merge = λt.λ(xr, yr).
   imap (λk.λc. t k → first c; second c) (zip xr yr)

disp = λm.λar.
   (rotate -m ar, rotate m ar)

----

L = λar. map (λa. cons a nil) ar

----

E = λp.λq.λar.
   merge (λk. (= (bit p k) (bit q k)))
         (zip ar (merge (λk. (= (bit p k) 1)
                        (disp (- 2^q 2^p) ar)))
```

```
I = λp.λar.
   merge (λk. (= (bit p k) 0)) (disp 2^i ar)

S = λf.λg.λp.λv.λar.
    merge (λk. (= (bit p k) v))
          (map f (zip (rotate -2^p ar) ar), map g ar)

S_id = λp.λv.λar.
   S (<>) (λx. nil) p v ar
```

The translation scheme from compositions of the bit-operations as arguments of `route` into compositions of `E`, `I`, and `S` is given below:

```
route f ≡ T[[f]]∘L

T[[e p q]]      =  E p q
T[[i p]]        =  I p
T[[s p v]]      =  S_id p v
T[[f∘g]]        =  T[[f]]∘T[[g]]
```

The `L` operation converts every element in an array into a singleton list. This is necessary because `route` returns an array of lists. The following lemmas will be useful for subsequent transformations which rely on optimizing communication forms:

Lemmas of Communication Operations

L2: $(map\ f)\circ(I\ p) \equiv (I\ p)\circ(map\ f)$
L3: $(map\ f)\circ(E\ p\ q) \equiv (E\ p\ q)\circ(map\ f)$

These lemmas indicate that it is permissible to promote computation into communication consisting of compositions of `I` and `E`. This is not surprising since `I` and `E` are bijections. The `S` operation is more powerful than either of these because it offers the ability to specify a combining operation to perform computation. Consider the general case of `map f` composed with S_{id}:

```
                        (map f)∘(S_id p v)

⇒ (map f)∘(S (<>) (λx. nil) p v)                          [unfold S_id]
⇒ (map f)∘(λar. merge (λk. (= (bit p k) v))
                      (map (<>) (zip (rotate -2^p ar) ar),
                       map (λx. nil) ar))                 [unfold S]
⇒ (map f)∘(λar.
     imap (λk.λc. (= (bit p k) v) → first c; second c)
          (zip (map (<>) (zip (rotate -2^p ar) ar))
               (map (λx. nil) ar)))                       [unfold merge]
⇒ λar.
     map f (imap (λk.λc. (= (bit p k) v) → first c; second c)
                 (zip (map (<>) (zip (rotate -2^p ar) ar))
                      (map (λx. nil) ar)))                [P1]
```

```
⇒ λar.
     imap (λk.λc. (= (bit p k) v) → f (first c); f (second c))
          (zip (map (<>) (zip (rotate -2^p ar) ar))
               (map (λx. nil) ar))                          P7 & P2
⇒ λar.
     imap (λk.λc. (= (bit p k) v) → first∘(f × id) c;
                                    second∘(id × f) c)
          (zip (map (<>) (zip (rotate -2^p ar) ar))
               (map (λx. nil) ar))                          P4 & P5
```

The goal of the transformation is to push the application of `f` through the body of the expression. The application of `f` is stuck inside the conditional statement and in order to lift it out, it is necessary to replace the occurrence of (`f × id`) and (`id × f`) with (`f × f`) so that both branches of the conditional contain the same sub-expression. The occurrence of (`f × f`) in both branches of the conditional can then be lifted out and mapped onto the array arguments. This changes the semantics of the program however, since arrays are strict data-structures. The strictness property of arrays in PAM is analogous to the notion that a lazy scalar machine is strict in all of the bits in the machine word-size, regardless of which degree of precision is required for the result.

As it stands, the function `f` is only applied to selected elements from the rotated or original array `ar`. The proposed transformation will apply `f` to all of the elements in both arrays. The transformation is only valid if applying `f` to all of the elements in both arrays does not cause an error. The qualification which makes this transformation valid is in the case where the function `f` is *bottom-reflecting*:[3]

definition: f is $\perp$-reflecting if $f\,x = \perp$ implies that $x = \perp$.

with this restriction in mind:

```
⇒ λar.
     imap (λk.λc. (= (bit p k) v) → first∘(f × f) c;
                                    second∘(f × f) c)
          (zip (map (<>) (zip (rotate -2^p ar) ar))
               (map (λx. nil) ar))                          iff f ⊥-reflecting
⇒ λar.
     imap (λk.λc. (= (bit p k) v) → first c; second c)
          (zip (map f∘(<>) (zip (rotate -2^p ar) ar))
               (map f∘(λx. nil) ar))          abstraction & P3 & property of zip
⇒ λar.
     imap (λk.λc. (= (bit p k) v) → first c; second c)
          (zip (map (λx.λy.f (x<>y))
                    (zip (rotate -2^p ar) ar))
               (map (λx. f nil) ar))                        property of <> & P1
```

[3]This is not a very serious limitation since in well-written programs most functions probably have this property. Functions which cause an error for valid inputs in their domain are not bottom-reflecting.

If `f` distributes through append, e.g. `f (x<>y) ≡ (f x) ⊗ (f y)` (where ⊗ is an associative function, dependent upon `f`) then it is possible to push the application of `f` further into the expression. Furthermore, assume that `f nil ≡ d` where `d` is a value in the range of `f`. Resuming the transformation yields:

```
⇒ λar.
    imap (λk.λc. (= (bit p k) v) → first c; second c)
         (zip (map (λx.λy.(f x) ⊗ (f y))
                   (zip (rotate -2^p ar) ar))
              (map (λx. d) ar))                    distr. prop. of f & defn. of f
⇒ λar.
    imap (λk.λc. (= (bit p k) v) → first c; second c)
         (zip (map (⊗) (zip ((map f) (rotate -2^p ar))
                            ((map f) ar)))         property of zip
              (map (λx. d) ar))
⇒ λar.
    imap (λk.λ(x, y). (= (bit p k) v) → x; y)
         (zip (map (⊗) (zip ((rotate -2^p) ((map f) ar)))
                            ((map f) ar)))         P9
              (map (λx. d) ar))
⇒ λar. let br = map f ar in
    imap (λk.λc. (= (bit p k) v) → first c; second c)
      (zip (map (⊗) (zip (rotate -2^p br) br)
           (map (λx. d) ar))                       abstraction
```

All that is required now is to change the last remaining occurrence of `ar` to `br`:

```
⇒ λar. let br = map f ar in
    imap (λk.λc. (= (bit p k) v) → first c; second c)
         (zip (map (⊗) (zip (rotate -2^p br) br)
              (map (λx. d)∘f ar))                  id
⇒ λar. let br = map f ar in
    imap (λk.λc. (= (bit p k) v) → first c; second c)
         (zip (map (⊗) (zip (rotate -2^p br) br)
              ((map (λx. d))∘(map f) ar))          P7
⇒ λar. let br = map f ar in
    imap (λk.λc. (= (bit p k) v) → first c; second c)
         (zip (map (⊗) (zip (rotate -2^p br) br)
              (map (λx. d) ((map f) ar)))          unfold ∘
⇒ λar. let br = map f ar in
    imap (λk.λc. (= (bit p k) v) → first c; second c)
         (zip (map (⊗) (zip (rotate -2^p br) br)
              (map (λx. d) br))                    by defn. of br
⇒ (λbr. imap (λk.λc. (= (bit p k) v) → first c; second c)
             (zip (map (⊗) (zip (rotate -2^p br) br)
                  (map (λx. d) br)))∘(λar. map f ar)   P3
⇒ (λbr. merge (λk. (= (bit p k) v))
              (map (⊗) (zip (rotate -2^p br) br),
               map (λx. d) br))∘(λar. map f ar)    fold with merge
```

```
⇒ (λf.λg.λp.λv.λar. merge (λk. (= (bit p k) v))
                          (map f (zip (rotate -2^p ar) ar),
                           map g ar)) ⊗ (λx. d) p v br)∘
   (λar. map f ar)                                        η-abstraction
⇒ (S ⊗ (λx. d) p v br)∘(λar. map f ar)                     fold with S
```

which has led to the proof of the following lemma:

L4: $(map\ f)\circ(S_{id}\ p\ v) \equiv (S \otimes (\lambda x.\ d)\ p\ v)\circ(map\ f)$

In order for compositions of `s` to be equivalent to `route`, `d` must be an identity of ⊗. The proof of this lemma imposed conditions on the distributive property of `f`, the associativity of ⊗, and the presence of an identity element, `d`. These conditions satisfy the definition of a *monoid*. A monoid is a triple $(A, \otimes, e)$ of objects A, an associative operation on these objects ⊗, and an identity of this operation, e. Lemma *L4* can be restated in terms of monoids and homomorphisms as: given two monoids M_1, M_2: $(\alpha, \oplus, e)$, $(\beta, \otimes, d)$ and a homorphism between them $f: \alpha \rightarrow \beta$, then:

$$(map\ f)\circ(S \oplus (\lambda x.\ e)\ p\ v) \equiv (S \otimes (\lambda x.\ d)\ p\ v)\circ(map\ f).$$

This lemma is extremely useful in that it shows how `map` can be promoted through the `s` operation by changing the function parameters appropriately.

For example, consider the following use of the `route` function:

```
sum = λnil. 0
    | λcons x xs. (+ x (sum xs))

sum∘(select 0)∘(route (λx. 0))
```

This expression directs all elements of an array to the first location. The accumulated list is then selected and summed. This expression has parallel complexity of $O(n)$, i.e. it is linear in the size of the array. This can be improved by the use of transformation. In the following example, it is assumed that the argument array has 8 elements:[4]

```
sum∘(select 0)∘(route (λx. 0))

⇒ sum∘(select 0)∘(route (s 2 0)∘(s 1 0)∘(s 0 0))          fn. equivalence
```

The starting point is to represent the routing function as a composition of the previously defined bit-operations. In this case, progressively setting all the bits of any source index to 0 is functionally equivalent to the original routing function `(λx. 0)`.

```
⇒ sum∘(select 0)∘((S_id 2 0)∘(S_id 1 0)∘(S_id 0 0)∘L)      translation
```

[4]Larger arrays can be accomodated by composing additional (`s`) bit-operations in `route`.

$\Rightarrow$ (select 0)∘(map sum)∘(S_{id} 2 0)∘(S_{id} 1 0)∘(S_{id} 0 0)∘L) [P10]

The sum function is a homomorphism from the monoid ($List_{\Re}$, <>, nil) to the monoid of real numbers ($\Re$, +, 0) because:

```
    sum nil ≡ 0
sum (x<>y) ≡ (sum x) + (sum y)
```

so therefore it is permissible to apply lemma *L4* to yield:

$\Rightarrow$ (select 0)∘(S (+) (λx. 0) 2 0)∘(map sum)∘ [L4]
(S_{id} 1 0)∘(S_{id} 0 0)∘L

$\Rightarrow$ (select 0)∘(S (+) (λx. 0) 2 0)∘(S (+) (λx. 0) 1 0)∘ [L4]
(S (+) (λx. 0) 0 0)∘(map sum)∘L

$\Rightarrow$ (select 0)∘(S (+) (λx. 0) 2 0)∘(S (+) (λx. 0) 1 0)∘
(S (+) (λx. 0) 0 0)∘(map id) [P7]

$\Rightarrow$ (select 0)∘(S (+) (λx. 0) 2 0)∘(S (+) (λx. 0) 1 0)∘ [P6]
(S (+) (λx. 0) 0 0)

The map sum function has been promoted through the composed S_{id} operations until it consumed the L operation at the front of the composed expressions. The result is a sequence of S operations with addition as the combining function. No intermediate list structures in the array are ever constructed. The resulting program has parallel complexity of O(*lg*(*n*)).

The range of routing functions described by the three bit-manipulation operations *i*, *e*, and *s* includes many commonly used interconnection patterns. The transformations presented in this section allow routing functions (specified as composition of these basic expressions) to be translated into equivalent forms defined using fast and efficient rotate operations. Once in this form, it becomes possible to perform additional transformations via the application of some of the lemmas derived in this section to yield more parallel/efficient final forms.

Transforming non-local communication into equivalent forms defined in terms of rotate means that resulting programs gain from the contention-free communication facilities exploited by this operation. Such gains are not inconsiderable because they make it possible to execute programs on architectures lacking (expensive) non-local communication facilities. A subset of functional data-parallel programs with all non-local communication operations removed (e.g. fetch, send, scatter, gather) and replaced by equivalent parallel data-transforms (E, I, and S) belong to a specific class of algorithms known as systolic algorithms which are discussed next.

7.2.5. Sample Application: Systolic Algorithm Derivation

Systolic algorithms have been found for a wide range of computationally-intensive problems found in scientific applications [Kung82]. Systolic algorithms are communication-efficient, scalable, massively-parallel algorithms typically consisting of a mesh of locally-interconnected simple processes. The communication links between processes are fixed. Inputs are fed in at the edge(s) of a systolic network and computation proceeds in a wavefront from the outer to the inner processes. Input values are propagated along with computed values and typically re-used at different points in the network. Output values are incrementally computed and propagated outwards towards an edge of the network. Computation is interleaved with communication so that the benefits of pipelining can be realized.

The regular structure, scalability, and fixed local interconnectivity of systolic algorithms make for highly efficient implementations and systolic algorithm design is an active area of research. Systolic algorithms can be implemented directly in custom hardware (VLSI), on dedicated programmable systolic machines, static-process parallel machines with limited interconnectivity, and make very good SIMD algorithms.

Existing systolic algorithms have traditionally been synthesized from imperative specifications. These formulations are inadequate because they over-specify the problem by imposing unnecessary ordering on computation. A set of monolithic array operations in a declarative context frees the specification of algorithms from these constraints. The data-parallel specification can be refined via standard transformational approaches to remove the need for non-local communication (e.g. by using the communication transformation lemmas developed in §7.2.4).

Development of systolic algorithms via program transformation is still an open research area. The advantages of using a functional language augmented with data-parallel operations are outlined below. It is beyond the scope of this thesis to develop such a methodology. Instead, a broad outline of where the monolithic, data-parallel style of programming coupled with an algebra for transforming communication and computation might lead to as a design methodology for systolic algorithms is presented.

7.2.5.1. Existing Approaches to Deriving Systolic Algorithms

The earliest methods for deriving systolic algorithms were completely ad-hoc and relied on the intuition of the algorithm designer. In an attempt to establish a formal basis for the derivation of provably correct systolic algorithms, techniques based on the techniques of dependence analysis used by vectorizing compilers for imperative languages were adopted (examples of this approach include [Quinto84] and [Capell83]). In a specific instance of this approach, the specification of an algorithm consists of a simple recurrence equation using array data-structures enclosed in nested Fortran DO-loops.

The problem with this approach is that it is very restrictive and generally only applicable to very simple algorithms. This is because imperative languages which program aggregates at the element-wise level suffer from a variety of defficiencies as detailed in Chapter 1. The shortcomings of these languages for data-parallel programming also preclude their effective use as specifications for the derivation of systolic algorithms (which are a sub-class of data-parallel algorithms).

7.2.5.2. Using Declarative Array Operations

The advantage of using operations without side-effects is that it is possible to use transformational methods to derive versions of algorithms which satisfy specific constraints (e.g. a systolic version of an algorithm initially specified using non-systolic operations). The number of dependencies is also reduced in programs written in a functional language. The formal semantics of functional languages means that it is possible to start from an easily-understood and inefficient high-level program and by applying a series of meaning-preserving transformations derive an equivalent (hopefully more efficient) systolic solution.

Interestingly, in our language it is possible to identify the systolic nature of an algorithm as a static property of the program form. That is, the set of data-parallel operations can be divided into systolic and non-systolic and the goal of transformation can be specified as: "derive a version of an algorithm equivalent to the original specification which does not use any non-systolic operations." The systolic primitives include `imap` and `rotate`. The `send` and `fetch` primitives are non-systolic. In addition, the properties of the primitive data-parallel operations can be used to promote computation into communication to gain additional parallelism.

The transformational approach to synthesizing systolic arrays starts from a monolithic data-parallel formulation of an algorithm. This forms the initial specification. The goal of the transformational approach is to transform this initial form into an equivalent systolic algorithm using the lemmas given previously (or additional lemmas derived along the way). The use of these meaning-preserving transformations guarantees that the transformed program remains equivalent to the original. The use of the data-parallel operations as a notation for expressing algorithms means that all intermediate programs remain fully executable.

Chapter 8

8. Further Work & Conclusions

In this chapter the previous chapters are summarized and areas for further research are presented. This includes exploring alternative data-allocation strategies which allocate data in accordance with how elements are accessed. Data-allocation strategies are closely related to the idea of load-balancing which consists of allocating multiple tasks to a single PE to achieve better overall PE utilization. An unexplored area is the integration of data-parallel I/O into a functional language framework. The conventional abstraction of I/O as operations on input and output streams is potentially useful as a model of data-parallel I/O.

The work in this thesis is compared and contrasted with relevant work in the literature. The literature survey traces the development of languages for the exploitation of data-parallelism starting from APL. Both functional and non-functional languages are presented and compared with regards to their support for monolithic operations on aggregate data-types, absence of a global state, enforcement of locality constraints, constructs to express communication, and freedom from side-effects.

8.1. Thesis Summary

The thesis starts from an exploration of the notion of dependencies and the implications for the exploitation of parallelism in Chapter 1. Two computational models: process- and data-parallel were identified together with corresponding machine architectures, MIMD and SIMD. The architectural assumptions of a von-Neumann machine underlying the design of sequential imperative languages were shown to preclude the effective exploitation of data-parallelism in these languages. The requirements for a language for data-parallelism were argued to include:

- Monolithic, higher-order, operations on aggregate data-types.
- Lack of a globally-accessible state.
- Strict enforcement of locality
- Constructs to express communication.
- Freedom from side-effects.

It was claimed that these requirements could be best met by a functional language augmented by a set of data-parallel operations on a primitive array data-type.

In Chapter 2 a simple functional language and an associated set of data-parallel operations on arrays were introduced. These primitive operations were shown to be general enough to permit a number of common data-parallel operations to be defined as derived operations.

The use of these operations for the purpose of programming sample applications was presented in Chapter 3. The sample applications were chosen from application domains which traditionally have not been suited to functional languages or data-parallel implementation. The primitive and derived operations permitted data-parallel programs to be written in a succinct manner consistent with the functional style.

In Chapter 4 an abstract data-parallel machine was introduced by specifying an abstract scalar machine for the compilation of functional languages, the SSTGM. Development of the data-parallel machine was explored in terms of the extensions required to the scalar machine. The abstract data-parallel machine, PAM, was shown to consist of a scalar machine augmented by an additional, planar, memory mode with basic instructions which apply to an entire plane of values at a time. Conditional execution was implemented by the addition of an activity mask to disable processing elements. The two instruction modes of PAM required that every function be compiled for both scalar and planar

execution. The compilation scheme translated all basic instructions and control constructs into equivalent forms for planar execution. The close correspondence between PAM and current SIMD machines was identified along with explanations of how both SIMD and MIMD machines could implement PAM.

The full specification of PAM in terms of its machine state and instructions was provided in Chapter 5. Following the specification of the compilation scheme, particular aspects of the compilation process regarding the implementation of particular language constructs were explained:

- Each user-defined function is compiled with both a scalar and a planar entry point.

- The `map` operation is compiled into a branch into the planar code stream.

- Expressions shared between scalar and planar code streams are reduced twice.

- Suspensions require that the context (consisting of the activity mask and the bounds register) be saved within the suspension when they are created and restored when they are evaluated.

- The execution of conditional statements relies on a register to keep track of the number of arguments on the stack. All continuations of a conditional statement are executed, which requires all the original arguments to be available to each continuation in turn, followed by an explicit combining operation to merge the partial results returned by each continuation into a single result.

- The presence of conditional statements affects the implementation of algebraic data-types because individual terms of sum-types are no longer mutually exclusive: a returned ADT object in the planar code-stream can consist of several instances of the same term of a data-type.

The execution of compiled program fragments was discussed in detail in Chapter 6. Three particular aspects were discussed in detail: recursion, higher-order functions, and algebraic data-types. These particular features are unique to the work in this thesis and therefore deserved particular attention. It was shown how the abstract machine model supports arbitrary recursive functions applied to planar-arguments. Since functions are first-class objects in our language, it is possible to program with "active" arrays, consisting

of arrays whose elements are functions. These are implemented as partially-applied planar functions. It is also possible to return arrays whose elements are algebraic data-types. The normal-order reduction strategy employed in this thesis permits the use of infinite array elements. Discussion of the run-time behavior of these code fragments led to a discussion of efficiency. A metric for evaluating the efficiency of particular programs executing on PAM was proposed. The source of inefficiency was attributed to the generated code, machine model, or the particular algorithm.

The discussion of the sources of inefficiency led directly to the subject of optimizations in Chapter 7. Optimizations were divided into two classes: automatic and non-automatic. The automatic optimizations could be realized by a compiler and some optimizations which could be applied to PAM consisted of adaptations of existing optimizations, such as strictness analysis and sharing analysis. Existing strictness analysis schemes need to be modified to take into account the semantics of conditional statements in the planar code-stream. Since all continuations are executed and arrays are strict data structures, more strictness information can be uncovered and there are fewer opportunities for lazyness which increases efficiency by reducing execution overheads. Normal-order reduction was shown to allow conservation of PE memory because of the demand-driven nature of computation under this scheme. New optimizations were discussed which arise from the single-instruction, multiple-datastream execution model employed by PAM. Conditional statements can give rise to bifurcating control flow and involve significant overheads due to the required testing of the activity mask and the saving of arguments. It was shown how these overheads could be reduced for specific code instances.

Non-automatic program optimizations require the intervention of the programmer and fall under the domain of program-transformation techniques. Two approaches to program transformation were discussed, the unfold/fold methodology and the algebraic methodology. These two styles of program-transformation were shown to complement each other in the development of an algebra for transforming data-parallel functional programs. The unfold/fold methodology was shown to be applicable to user-defined functions whereas the algebraic approach was suited to transforming expressions comprising data-parallel primitives. The algebraic system relied on a set of axioms and lemmas. Some initial axioms of the data-parallel primitives were given and used in later transformation examples.

The development of an algebra for the transformation of communication was derived from the work on Parallel Data Transforms and augmented by the addition of an extra operation to permit the specification of more general routings. An equivalence between general, non-local forms of communication and efficient local forms was expressed as a translation scheme. The efficient local forms of communication were defined in terms of compositions of `rotate` operations which allowed computation to be

promoted into communication. The resulting techniques were used to illustrate how initial, sequential, specifications could be transformed by promoting computation to yield parallel and efficient program forms.

8.2. Further Work

A number of issues remain to be addressed. The ground-work for the development of efficient and parallel algorithms by the use of the transformational methods in Chapter 7 has been laid but the usefulness of such a technique in specific application domains remains to be explored. Some specific issues which remain open research areas are discussed in the following sections.

8.2.1. Non-Canonical Data-Allocation

Recent papers exploring optimizations of programs on current SIMD machines (e.g. Reddaway's work on improving programs on the AMT DAP [Reddaw89], Knobe et. al. on the efficiency of different allocation schemes on the Connection Machine to reduce the need for communication [Knobe90], and Flanders' work on PDTs indicate that the allocation of data-elements to PEs is critical to performance.

In this thesis a canonical allocation has been assumed whereby array elements with the same index are allocated to the same PE. This does not take into account the way in which elements are combined during the course of computation. Ideally, array elements should be allocated on the basis of *where* they will be used, with a view to minimizing the need for communication. Reddaway discusses the advantages of *crinkle* versus *sheet* array-allocation schemes in the context of particular applications on the DAP. Knobe discusses some particularly effective heuristics for allocating *array sections* in Fortran-90. The incorporation of similar techniques into the work in this thesis is clearly desirable. This could possibly take the form of an analysis technique which partially-evaluates expressions consisting of communication operations at compile-time in order to minimize the need for intermediate communication.

8.2.2. Load-Balancing

Data-allocation strategies introduce the possibility of allocating multiple data-elements to a single PE in order to eliminate communication between adjacent elements (at the expense of added computation at each PE). This leads into the area of load-balancing. Resolving the cardinality variation between data-elements and PEs involves decisions which take into account the relative cost of communication and computation for specific algorithms and target machines. For tasks where the cost of computation is known various algorithms for effective load-balancing exist (see [Kruatr88], [Berman87]). In general, the cost of computation depends on run-time values which determine the flow of control.

PAM assumes an infinite number of available PEs since planes can be of any size. The emulation of this infinite machine model (presented in Chapter 4) assumes that planes which exceed the physical machine size are implemented as tiles. Computation applied to planes is applied to all of the tiles of a plane in turn. This particular model does not permit individual PEs to proceed to the corresponding element in the next tile until all PEs have finished. The Connection Machine supports *virtual PEs* where multiple virtual PEs are emulated by a single physical PE which corresponds to the MIMD notion of multi-tasking (i.e. instantiating concurrent processes on a single processor) [Thinki87]. The MasPar MP-1 supports indirect-addressing which can be used to yield the same effect because tiles may be allocated to contiguous addresses in PE memory so that an individual PE can reference an element in successive tiles as consecutive offsets from a base-address. For particular applications, this data-parallel form of multi-tasking can help to raise processor-utilization by allocating multiple instances of the same task to individual PEs [Tombou89] and permitting each PE to work on the next element as soon as it completes work on the current one. This technique is particularly effective in the case where the execution time of a task is dependent upon the value of its inputs. Previously, PEs with short task times would render themselves inactive for the remainder of the computation whereas under a load-balancing scheme these PEs could proceed with executing the same task on the next data-element in their allocation.

Any scheme which clusters data to improve load-balancing may produce detrimental effects when communication of data-elements is required. The improvement in PE utilization derived by load-balancing may be offset by the increased overheads in the communication of data-elements since each PE is responsible for several data-elements.

8.2.3. Parallel I/O

Support for input and output in data-parallel computation was not addressed in the thesis. To avoid computation being I/O-bound, support must be provided for parallel I/O. In functional languages, I/O is typically modelled by the abstraction of a stream (lazy list) of input and output. Input is performed by removing elements from the input stream, output by appending them to the output stream. By abstracting input and output as operations on a data-structure, the side-effect free semantics can be preserved. In the case of a data-parallel model of computation, it is possible use a similar abstraction of I/O. A variant of the `shift` operation can be provided which takes as additional arguments the input and output streams, which has the following type:

```
ioshift : index × array(index, α) × list α × list α →
          array(index, α) × list α × list α
```

The first list argument is the input stream, the second the output stream. To perform parallel I/O, these "streams" must permit parallel access to input and output

elements. The advantage of this abstraction of I/O is that the internal representation of these (parallel) streams is completely unspecified since the streams are not directly manipulated by the programmer: access to I/O is always indirect, via the use of operations such as `ioshift`. The values at the "edge" of the array which are vacated by the `shift` operation are filled (in parallel) by elements from the input stream. Similarly, those elements which are shifted out of the array are placed on the output stream. The function returns the shifted array and the new input and output streams. It is also possible to define a variant of the `send` operation where each element on the input stream has an associated destination index. In this case, elements in the input stream can be routed to their destinations directly without first shifting them into an array.

Parallel I/O is critical to sustained performance in real-word applications and although this thesis does not address the subject it is obvious that parallel forms of I/O will by necessity be monolithic in style. As a result, operations for parallel I/O should be consistent with the monolithic communication operations provided to support inter-PE communication.

8.3. Review of Related Work

The origin of the monolithic approach to programming with aggregate data structures dates back to Iverson's language APL [Iverso62]. APL includes a number of first- and second-order operations which can be used to perform a variety of common mathematical operations on arrays. This style of programming was adopted by the functional language community with the introduction of Backus' FP language which relied on a set of pre-defined second-order operations defined on *sequences* (heterogeneous lists) [Backus78]. FP is not fully higher-order nor does it support any additional types of data-structures. Nevertheless, the simplicity of FP has led a variety of researchers to consider implementing FP on vector-processing machines (see [Budd88]). Sequences are unsuitable aggregates for data-parallel computation because they are heterogeneous and used in conjunction with list manipulation operations (e.g. `hd`, `tl`, and `append`) which do not permit a natural exploitation of data-parallelism. An implementation of FP's *apply-to-all*, *insert*, and some of its data-copying operations (e.g. *distl*, *distr*) using a data-parallel model of computation has been developed [Walins90]. This implementation imposes the restriction that all elements in a sequence have to be the same size and disallows the use of recursive functions as arguments to the *apply-to-all* (i.e. `map`) function. This system also requires a complex analysis technique called *structure inference* to infer how data is permuted by routing functions.

Bird & Meertens have developed a small-set of higher-order operations (scans, reduces, etc.) to explore the development of algorithms which exploit data-parallelism by algebraic transformation of initial specifications [Bird87], [Meerte89]. These initial

specifications take the form of algorithms expressed using their list-based monolithic operations. The absence of side-effects in these operations means that such transformations take the form of algebraic laws which are guaranteed to preserve meaning. Skillicorn has shown that such operations can be used effectively across a wide range of parallel architectures including MIMD architectures [Skilli90].

These early languages were not developed for their ability to express parallel computation. Designed for sequential implementation, they continue to reflect the inherent assumptions of a globally-accessible and uniform state. This is evident in the fact that they fail to provide effective primitives to describe communication and do not enforce locality constraints.

A particular class of algorithms based on the scalability and efficiency of computation which relies only on local interconnectivity between processing elements include so-called *systolic algorithms*, a term coined by Kung (see [Kung82]). These algorithms are particularly amenable to direct realization in VLSI or as efficient program forms for data-parallel computers. A number of methods based on the analysis of data-dependencies and the projection of multi-dimensional integer convex sets have been developed for the derivation of systolic algorithms. Initial specifications consist of imperative code fragments which use element-wise operations on arrays. The earliest examples of this type of analysis includes the work by Quinton, Capello, and Steiglitz [Quinto84], [Capell83]. This method of deriving data-parallel algorithms differs from that of the previously mentioned approaches in that an element-wise specification in an imperative language is used. The work required for the derivation of a parallel systolic algorithm from these initial specifications is very similar to that attempted by automatically vectorizing compilers. So far, such methods have been hampered by lack of generality: a design method for extracting systolic algorithms from arbitrary imperative code fragments does not exist. Recognition of the limitations in an element-wise problem specification for the exploitation of data-parallelism motivates the search for languages based on monolithic methods.

The simplicity and scalability of data-parallel computation suggests its utility as a general model for distributing computation in parallel machines. Jesshope proposes a Virtual System Architecture (VSA), a general data-parallel model of computation for MIMD and SIMD computers based on user-defined, parallel data-structures [Jessho90]. The VSA model is based on the notions of selection and the application of concurrent operations on subsets of distributed data. The VSA model is a more general abstract machine for data-parallel computation than PAM. It is not a language for data-parallel programming but a high-level intermediate form designed to support parallel languages. An example includes a new language, EVAL, developed to aid the expression of vector-parallelism [Muchni91].

These previous approaches stem from a "language-first" (top-down) design approach. The "machine-first" philosophy (bottom-up) has led to a number of specific languages created for the exploitation of specific data-parallel machine features. Such languages include MPL [MasPar90], Fortran-Plus [Jenkin90], DAPL [Rice89], Parallaxis [Br91], CM-Lisp and it's lower-level variant, *Lisp [Thinki87].

These languages can be divided into two groups: (1) those languages that overload basic arithmetic/logical operations on arrays plus some dedicated (first-order) reduction and data-movement operations, (2) those languages that introduce higher-order operations to provide parameterized monolithic operations.

The first category includes MPL and Fortran-Plus which are extensions of C and Fortran respectively with support for SIMD machine features provided through libraries of procedures to support reduction and data-movement. As a result of overloading the basic arithmetic/logical and control operations to provide monolithic, data-parallel variants, the semantics of these languages become confused (there are two "readings" to every expression: sequential and parallel, and the sequential and parallel parts of a program are not clearly delineated). Also, because these languages are first-order, a great number of monolithic operations need to be provided which hampers accessability to data-parallelism. These languages do not present a clean, consistent, interface to data-parallelism.

The second category includes the CM-Lisp and *Lisp languages which are designed primarily to support the hardware features of the Connection Machine. CM-Lisp is a higher-level language than *Lisp. In CM-Lisp the presence of communication is made more explicit and second-order monolithic computation and data-movement operations are provided. The DAPL language is an attempt to introduce object-oriented features in an abstraction of a data-parallel machine. A substantial amount of support is provided for the programmer to define local and global forms of interconnection patterns using a set of pre-defined topologies (grids, binary trees, hypercubes, etc.) and to name common patterns of communication (up, down, north, south, east, west, etc.). In this it is similar to the approach adopted by the designers of Parallaxis which includes analogous but slightly more general features since topologies can be completely user-defined. These languages are restricted because they can only define static interconnections between PEs. They also support a *microscopic* model of data-parallel machines. The programming model in these languages is that of an "array-of-processors" rather than a "processor-of-arrays." The programmer writes programs from the perspective of a single processing element which enforces a strict notion of locality as only values in the local state are accessible. The monolithic approach presents a *macroscopic* approach where an operation applies to all processing elements in the machine. Interestingly, the introduction of higher-order monolithic operations provides a bridge between the microscopic and macroscopic models because the monolithic operation has a macroscopic semantics but the function argument

of the higher-order operations have a microscopic semantics (i.e. operate on single scalar values in local PE memory). The role of the compilation system is to translate microscopic functions to macroscopic equivalents in PAM-code.

None of these languages are free from side-effects and do not provide for the rich support of data-types found in functional languages. The presence of assignment means that it is much more difficult to provide an algebra for the transformation of programs.

The first category of machine-oriented languages are unsatisfactory because it is difficult to integrate monolithic features in element-oriented von-Neumann languages. The second category of languages attempts to redress this imbalance but in so doing defines new types of imperative programming paradigms which are unfamiliar and require new ways of thinking about programming.

A number of general models for the expression of parallel computation have attracted particular attention in the literature recently. These include, Carriero & Gelernter's Linda model [Carrie86], Chandy & Misra's UNITY model [Chandy88], and Sabot's work on the Paralation model [Sabot88] (which most resembles the work in this thesis).

Linda is a parallel-programming model which relies on a common content-addressable data-structure called *tuple-space* through which distinct processes interact by placing tuples of values in—and attempting to match values from—tuple-space. A process attempting to match a tuple not in the tuple-space suspends until a matching tuple becomes available. By medium of tuple-space and the suspending nature of match requests, a data-flow synchronization ordering is imposed on processes which communicate through tuple-space. The generality of a fully associative method of interaction allows for the expression of data-parallel style computation in Linda. Array elements can be placed into tuple-space and any number of instances of a function can be created to consume array elements and place the results back into tuple-space. The main drawback to this approach is that Linda purposely abstracts away two aspects of parallel programming with which the work in this thesis is most concerned: communication and locality. Linda programs are not transparent because of the asynchronous computational model and the unpredictable performance of tuple-space communication. Furthermore, it is not clear how the Linda model can be mapped on to more restrictive data-parallel architectures (e.g. SIMD).

A UNITY program is a series of statements that assign to variables where the statements are executed infinitely often. Computation is said to terminate when a program state reaches a fixed-point (i.e. no longer changes). UNITY difers from the work in this thesis in a number of ways. Primarily, UNITY is both a language and a proof system which allows the programmer to prove assertions about program behaviour. It is a fundamentally non-deterministic model of computation. Communication and locality are not explicit aspects of UNITY programs.

Sabot presents a computational model based on the notion of *Paralations* (parallel relations): data-parallel computation on aggregate data-structures called *fields* of indeterminate structure, where each element is associated with a unique index. The paralation model consists of three operations: `elwise`, `match`, and `move`. The `elwise` operation is a generalized form of higher-order monolithic operation used to express data-parallel computation. The `match` operation is used to compute a *mapping*, which is an interconnection pattern (allowing arbitrary fan-in and fan-out) between fields. The `match` operation takes two fields as arguments and compares field elements and connects elements in the source field to elements in the destination field if they contain the same value. The resulting interconnection pattern may be used to perform communication via the `move` (<-) operation. The `move` operation takes an interconnection pattern, a source and destination field, and moves elements from the source to the destination field. Elements sent to the same destination can be combined by a dyadic combining function specified as an optional parameter to `move`. The `move` operation therefore performs both communication and computation.

In an implementation of the Paralation model, PARALATION LISP, fields can be nested and side-effects may be carried out in an `elwise` operation. This allows the expression of parallel computation which is unintentionally non-deterministic and whose result is undefined. The problems introduced by variable aliasing means that such conditions cannot in general be detected by the compiler (nor, sometimes, by the programmer!). In addition, permitting side-effects means that arbitrary synchronization is required (Sabot indicates that machines should synchronize after every `elwise`) and that program transformation techniques become difficult to apply. These particular shortcomings are indictments of the implementation of the Paralation model in a language (Lisp) which permits side-effects and are not criticisms of the Paralation model itself.

Nevertheless, the Paralation model does not distinguish between local and global forms of communication despite the great difference in efficiency between these two forms. The Parallel Data Transform work involving the transformation of global communication into more efficient, local, equivalents cannot be used in this model. The Paralation model also loses transparency because its method of establishing interconnections for general communication is based on matching the contents of field elements. In our `fetch` and `send` operations in contrast, the interconnection pattern is specified in terms of destination/source index values. In our primitive operations, by separating global communication into two distinct operations, the programmer is made aware of the problems of communication involving arbitrary fan-in and fan-out: a `send` may involve arbitrary fan-in at the receiver whereas a `fetch` may involve arbitrary fan-out at the sender. The all-in-one `match` operation blurs these two distinct forms of global communication. More seriously, however, `match` needs to perform content-matching on

the values of field elements to set up interconnection patterns which requires a sorting operation on the two fields followed by a parallel-prefix operation to perform the comparisons. Since `match` is the only way of establishing interconnections, this cost is incurred for every new interconnection pattern. In many algorithms the interconnectivity can be computed directly in terms of the indices of array elements (e.g. fast-Fourier transform, perfect shuffle, etc.). Nevertheless, the Paralation model always requires the programmer to incur the cost of general matching of field values. Using our primitives, in contrast, the programmer is free to compute interconnection patterns algorithmically and the cost of doing so is clear. If a `match`-like facility is required, this can be achieved as a derived operation in terms of the primitive operations given. The true cost of `match` will then be made apparent. In the Paralation model no separate communication facilities are provided to express local communication. This is due to the fact that the Paralation model does not specify the physical adjacency of elements whose indices are logically adjacent.

The base language of the Paralation model, Lisp, is significantly simpler than the fully lazy functional language presented in this thesis. Lisp lacks lazy evaluation and algebraic data-types. The support of such features in the compilation scheme presented in this thesis yields a language with greater expressive power and which because it is fully functional (free from side effects) is amenable to transformation. The operations in this thesis present a more realistic abstraction of inter-PE communication and as a result our language is more natural for the exploitation of data-parallelism.

8.4. Conclusions

The contribution of this thesis has been to show that a functional language endowed with higher-order data-parallel operations is a natural vehicle for the exploitation of data-parallelism. This is a surprising result considering the almost total absence of research in adopting a data-parallel model of computation for functional languages. Programs written in our language are clear and concise. The monolithic style of programming is fully consistent with the functional one.

The choice of the Spineless-Tagless G-Machine as a starting point for the development of an abstract data-parallel machine was rewarded by the ease with which it was possible to accommodate the two different instruction streams. The uniformity of objects in the heap (i.e. all objects are closures) made it possible to accommodate planar algebraic data-types and the insertion of *merge* suspensions without any changes to the mechanics of the underlying abstract machine model. In an implementation of PAM this flexibility of the execution model may be invaluable as changes and modifications can be introduced simply by changing the code field of closures. New types of closures can be introduced without changing any other part of the compilation or run-time system.

The STGM made it easy to pursue a normal-order reduction strategy but does not require it. The compilation scheme in Chapter 5 is easily modified if an applicative-order reduction strategy is desired. The strict nature of basic planar operations coupled with a normal-order reduction strategy for user-defined functions makes for an evaluation order which is a hybrid between normal- and applicative-reduction. Only experience will tell whether this aids or detracts from the ease of programming. In addition, the space-benefits of normal-order reduction need to be weighed against the time-overheads incurred in supporting suspensions.

Transparency and its derivative concepts, uniformity and locality, can be preserved to yield predictable performance in a functional language exploiting a data-parallel model of computation. The freedom from side-effects in our language made it possible to adapt existing transformation methods to improve data-parallel programs. Transformations to exploit parallelism are easier in a data-parallel model than a process-parallel one because the creation of processes and the allocation of tasks and resources in the machine are all implicit in process-parallel implementations of functional languages: the programmer is hampered by the lack of transparency which could otherwise be used to guide transformation tactics. In contrast, the transformations developed in this thesis rely on the predictable behavior of the data-parallel operations and make the benefits of transformation readily apparent by inspection. Transformational methods based on the optimization of computation and communication have tremendous potential for the derivation of high-performance data-parallel algorithms from initial specifications.

The difficulties in extending imperative languages to the task of programming data-parallel machines have been apparent for some time. The relative lack of commercial success of data-parallel machines despite their long availability and high-performance may be due in some part to the perceived "difficulty" in programming these architectures. The problems lie in the von-Neumann assumptions built in to the semantics of sequential imperative languages. What is required is a new approach to the design of languages for data-parallelism. This thesis presented the argument in favor of the functional approach by capitalizing on the traditional strengths of functional languages and by showing how such a language benefits from a data-parallel model of computation.

REFERENCES

[Aasa88] Aasa, A., Holmström, S., and Nilsson, C. "An Efficiency Comparison of some Representations of Purely Functional Arrays." *BIT* **28**, 1988, pp. 490-503.

[Active88] *DAP Series: Technical Overview*, Active Memory Technology Ltd., 65 Suttons Park Avenue, Reading, RG6 1AZ, U.K., 1988.

[Ariola89] Ariola, Z. and Arvind, "P-TAC: A Parallel Intermediate Language." In *Functional Programming Languages and Computer Architecture,* September 1989, pp. 230-242.

[Arvind86] Arvind, , Nikhil, R.S., and Pingali, K.K. "I-Structures: Data Structures for Parallel Computing." Tech. Rept. 269, Computation Structures Group Memo, Massachusetts Institute of Technology, 545 Technology Square, Cambridge, MA 02139, U.S.A., February, 1986.

[August89] Augustsson, L. and Johnsson, T. "Parallel Graph Reduction with the <v, G>-Machine." In *Functional Programming Languages and Computer Architecture,* ACM, September 1989, pp. 202-213.

[Backho89] Backhouse, R. "An Exploration of the Bird-Meertens Formalism." In *STOP Summer School on Constructive Algorithmics,* September 1989, pp. 1-61.

[Backus78] Backus, J. "Can Programming be Liberated from the von-Neumann Style? A Functional Style and its Algebra of Programs." *Communications of the ACM* **21**, 8, August 1978, pp. 613-641.

[Bailey85] Bailey, R. "FP/M Abstract Syntax Description." Tech. Rept. Dept. of Computing, Imperial College, London SW7 2BZ, U.K., 1985.

[Barnsl88] Barnsley, M.F. and Sloan, A.D. "A Better Way to Compress Images." *Byte* , January 1988, pp. 215-223.

[Berman87] Berman, F. and Snyder, L. "On Mapping Parallel Algorithms into Parallel Architectures ." *Journal of Parallel and Distributed Computing* **4**, 5, October 1987, pp. 439-458.

[Bernec89] Bernecky, R. "Fortran 90 Arrays." *ACM SIGPLAN Notices* **26**, 2, February 1989, pp. 83-98.

[Bird87] Bird, R.S. "An Introduction to the Theory of Lists." In *Logic of Programming and Calculi of Discrete Design,* Broy, M. (Ed.), Springer-Verlag, 1987, pp. 5-42.

[Blank90] Blank, T. "The MasPar MP-1 Architecture." Tech. Rept. MasPar Computer Corporation, 749 North Mary Avenue, Sunnyvale, CA 94086, 1990.

[Blello89] Blelloch, G. "Scans as Primitive Parallel Operations." *IEEE Transactions on Computers* **38**, 11, November 1989, pp. 1526-1538.

[Blello90] Blelloch, G.E. and Sabot, G.W. "Compiling Collection-Oriented Languages onto Massively Parallel Computers." *Journal of Parallel and Distributed Computing* **8**, 2, February 1990, pp. 119-134.

[Bloss89] Bloss, A. *Path Analysis and the Optimization of Non-strict Functional Languages*, Ph.D. dissertation, Yale University, May 1989.

[Boug91] Bougé, L. "On the Semantics of Languages for Massively Parallel SIMD Architectures." In *Parallel Architectures and Languages Europe,* 1991, pp. (to appear).

[Br91] Bräunl, T. "Massively Parallel Programming with Parallaxis." Tech. Rept. Universität Stuttgart, Institut für Informatik, Azenbergstr. 12, D-7000 Stuttgart 1, FRG, 1991.

[Budd88] Budd, T.A. "Composition and Compilation in Functional Programming Languages." Tech. Rept. 88-60-14, Computer Science Department, Oregon State University, Corvallis, OR 97331, U.S.A., June, 1988.

[Burke86] Burke, M. and Cytron, R. "Interprocedural Dependence Analysis and Parallelization." In *SIGPLAN 86 Symposium on Compiler Construction,* July 1986, pp. 162-175.

[Burn87] Burn, G.L. "Evaluation Transformers – A Model for the Parallel Evaluation of Functional Languages (Extended Abstract)." In *Functional Programming Languages and Computer Architecture,* Kahn, G. (Ed.), Springer-Verlag, September 1987, pp. 446-470.

[Burn90] Burn, G.L. "Implementing Lazy Functional Languages on Parallel Architectures." In *Parallel Computers: Object-Oriented, Functional, Logic.* John Wiley, Treleaven, P.C. (Ed.), pp. 101-139, 1990.

[Bursta77] Burstall, R.M. and Darlington, J. "A Transformation System for Developing Recursive Programs." *Journal of the ACM* **24**, 1, 1977, pp. 44-67.

[Capell83] Capello, P.R. and Steiglitz, K. "Unifying VLSI Array Designs with Geometric Transformations." In *International Conference on Parallel Processing,* 1983, pp. 448-457.

[Carmic85] Carmichael, J.W.S. "History of the ICL Content-Addressable File Store (CAFS)." *ICL Technical Journal* **4**, 4, November 1985, pp. 352-357.

[Carrie86] Carriero, N., Gelernter, D., and Leichter, J. "Distributed Data Structures in Linda." In *Principles of Programming Languages,* ACM, St. Petersburg, Florida, January 1986, pp. 236-242.

[Chandy88] Chandy, K.M. and Misra, J. *Parallel Program Design: a Foundation,* Addison-Wesley 1988.

[Clack85] Clack, C. and Peyton-Jones, S.L. "Strictness Analysis–A Practical Approach." In *Functional Programming Languages and Computer Architecture,* Springer-Verlag, 1985, pp. 35-49.

[Collin88] Collins, J.F., Coulson, A.F.W., and Lyall, A. "The Significance of Protein Sequence Similarities." *CABIOS* **4**, 1, 1988, pp. 67-71.

[Cripps87] Cripps, M.D., Darlington, J., Field, A.J., Harrison, P.G., and Reeve, M.J. "The Design and Implementation of Alice: A Parallel Graph Reduction Machine." In *Selected Reprints on Dataflow and Reduction Architectures,* Thakkar, S. (Ed.), IEEE, 1987, pp. 1-22.

[Fairba87] Fairbairn, J. and Wray, S. "TIM: A Simple, Lazy Abstract Machine to Execute Supercombinators." In *Functional Programming Languages and Computer Architecture,* Kahn, G. (Ed.), Springer-Verlag, September 1987, pp. 34-45.

[Field88] Field, A.J. and Harrison, P.G. *Functional Programming,* Addison Wesley, International Computer Science Series 1988.

[Flande87] Flanders, P.M. and Parkinson, D. "Data Mapping and Routing for Highly Parallel Processor Arrays." *Future Computing Systems* **2**, 2, 1987, pp. 183-224.

[Flynn72] Flynn, M.J. "Some Computer Organizations and their Effectiveness." *IEEE Transactions on Computers* **c-30**, 7, September 1972, pp. 948-960.

[Gaudio85] Gaudiot, J.L. and Ercegovac, M.D. "Performance Evaluation of a Simulated Data-Flow Computer with Low-Resolution Actors." *Journal of Parallel and Distributed Computing* **2**, 1985, pp. 321-351.

[Gelern90] Gelernter, D. "Information Management in Linda." In *Parallel Processing and Artificial Intelligence,* Reeve, M. and Zenith, S.E. (Eds.), John Wiley & Sons, 1990.

[Glaser88] Glaser, H., Reeve, M., and Wright, S. "An Analysis of Reference Count Garbage Collection Schemes for Declarative Languages." Tech. Rept. Imperial College, Department of Computing, 180 Queens Gate, London SW7 2BZ, U.K., 1988.

[Glauer87] Glauert, J.R.W., Kennaway, J.R., and Sleep, M.R. "Dactl: A Computational Model and Compiler Target Language Based on Graph Reduction." *ICL Technical Journal*, May 1987, pp. 509-537.

[Hillis87] Hillis, W.D. "The Connection Machine." *Scientific American* **256**, 6, June 1987, pp. 108-115.

[Hudak85] Hudak, P. and Goldberg, B. "Distributed Execution of Functional Programs Using Serial Combinators." *IEEE Transactions on Computers* **c-34**, 10, October 1985, pp. 881-891.

[Hudak86] Hudak, P. and Young, J. "Higher-order Strictness Analysis in Untyped Lambda Calculus." In *Principles of Programming Languages,* 1986, pp. 97-109.

[Hudak88] Hudak, P. and Mohr, E. "Graphinators and the Duality of SIMD and MIMD." In *Lisp and Functional Programming,* ACM, July 1988, pp. 224-234.

[Hudak89] Hudak, P. "Conception, Evolution, and Application of Functional Programming Languages." *Computing Surveys* **21**, 3, September 1989, pp. 359-411.

[Hudak90] Hudak, P., Wadler, P., Arvind, , Boutel, B., Fairbairn, J., Fasel, J., Hughes, J., Johnsson, T., Kieburtz, D., Nikhil, R., Jones, S.P., Reeve, M., Wise, D., and Young, J. "Report on the Programming Language Haskell." Tech. Rept. 1.0, Language Specification, Yale University, April, 1990.

[Hughes87] Hughes, J. "Managing Reduction Graphs with Reference Counts." Tech. Rept. CSC/87/R2, Departmental Research Report, University of Glasgow, Department of Computing Science, Lillybank Gardens, Glasgow, G12 8QQ, U.K., March, 1987.

[ISO89] *Fortran 88: A Proposed Revision of Fortran 77*, ISO/IEC JTC1/SC22/WG5-N357, March, 1989.

[Iverso62] Iverson, K.E. *A Programming Language*, John Wiley & Sons, New York 1962.

[Jenkin90] Jenkins, H., *FORTRAN-PLUS Enhanced (FORTRAN*)*, Active Memory Technology, 65 Suttons Park Avenue, Reading RG6 1AZ, U.K., 1990.

[Jessho80] Jesshope, C.R. "The Implementation of Fast Radix 2 Transforms on Array Processors." *IEEE Transactions on Computers* **c-29**, 1, 1980, pp. 20-27.

[Jessho88] Jesshope, C.R., Miller, P., and Yantchev, J. "Programming with Active Data." In *PARCELLA*, Wolf, G., Legendi, T., and Schendel, U. (Eds.), Springer-Verlag, 1988, pp. 111-129.

[Jessho90] Jesshope, C. "The VSA: An Abstract Definition and Interface for Data-Parallel Program Generation." *Computers and Artificial Intelligence* **9**, 5, 1990, pp. 441-459.

[Johnss84] Johnsson, T. "Efficient Compilation of Lazy Evaluation." *SIGPLAN Notices* **19**, 6, June 1984, pp. 58-69.

[Jouret91a] Jouret, G.K. "Compiling Functional Languages for SIMD Architectures." In *Symposium on Parallel and Distributed Processing*, IEEE, December 1991, pp. (to appear).

[Jouret91b] Jouret, G.K. "A Foundation for Declarative Data-Parallelism." In *Implementation of Functional Languages on Parallel Architectures*, Glaser, H. and Hartel, P. (Eds.), University of Southampton, Department of Electronics and Comp uter Science, June 1991, pp. 311-330.

[Knobe90] Knobe, K., Lukas, J.D., and Steele, G.L. "Data Optimization: Allocation of Arrays to Reduce Communication on SIMD Machines." *Journal of Parallel and Distributed Computing* **8**, 1990, pp. 102-118.

[Kruatr88] Kruatrachue, B. and Lewis, T. "Grain Size Determination for Parallel Processing." *IEEE Software* **5**, 1, January 1988, pp. 23-32.

[Kung82] Kung, H.T. "Why Systolic Architectures?." *Computer* **15**, 1, January 1982, pp. 37-46.

[Lampor74] Lamport, L. "The Parallel Execution of DO Loops." *Communications of the ACM* **17**, 2, February 1974, pp. 83-93.

[Landin64] Landin, P.J. "The Mechanical Evaluation of Expressions." *Computer Journal* **6**, 1964, pp. 308-320.

[Leicht89] Leichter, J.S. "Shared Tuple Memories, Shared Memories, Buses and LAN's–Linda Implementations Across the Spectrum of Connectivity." Tech. Rept. YALEU/DCS/TR-714, Yale University, July, 1989.

[Malcol89] Malcolm, G. "Homomorphisms and Promotability." In *Mathematics of Program Construction,* van de Snepscheut, J.L.A. (Ed.), Springer-Verlag, 1989, pp. 335-347.

[Marino89] Marino, G. and Succi, G. "Data Structures for Parallel Execution of Functional Languages." In *Parallel Architectures and Languages Europe,* Odijk, E., Rem, M., and Syre, J.C. (Eds.), Springer-Verlag, 1989, pp. 346-356.

[MasPar90] *MasPar Parallel Application Language (MPL),* MasPar Computer Corporation, 749 North Mary Avenue, Sunneyvale, CA 94086, PN 9302-0000-2790, 1990.

[Meerte86] Meertens, L. "Algorithmics–Towards Programming as a Mathematical Activity." In *Mathematics and Computer Science,* de Bakker, J.W. and Hazewinkel, M. (Eds.), North Holland, 1986, pp. 289-334.

[Meerte89] Meertens, L. "Constructing a Calculus of Programs." In *Mathematics of Program Construction,* van de Snepscheut, J.L.A. (Ed.), Springer-Verlag, 1989, pp. 66-90.

[Monro90] Monro, D.M., Dudbridge, F., and Wilson, A. "Deterministic Rendering of Self-Affine Fractals." In *Colloqium on the Application of Fractal Techniques in Image Processing,* IEE, December 1990.

[Muchni91] Muchnik, V.B. and Shafarenko, A.V. *The Language EVAL and its Implementation,* Pitman, Research Monographs in Parallel and Distributed Computing 1991.

[O85] O'Donnell, J.T. "An Architecture that Efficiently Updates Associative Aggregates in Applicative Programming Languages." In *Functional Programming Languages and Computer Architecture,* Springer-Verlag, 1985, pp. 164-189.

[Patel85] Patel, D., Schlag, M., and Ercegovac, M. "vFP: An Environment for the Multi-Level Specification, Analysis, and Synthesis of Hardware Algorithms." In *Functional Programming Languages and Computer Architecture,* Springer-Verlag, September 1985, pp. 238-255.

[Perry88] Perry, N. "Hope+." Tech. Rept. IC/FPR/LANG/2.5.1/7, Department of Computing, Imperial College of Science, Technology and Medicine, London, U.K., February, 1988.

[Peyton87a] Peyton-Jones, S.L., Clack, C., Salkild, J., and Hardie, M. "GRIP–A High-Performance Architecture for Parallel Graph Reduction." In *Functional Programming Languages and Computer Architecture,* Kahn, G. (Ed.), Springer-Verlag, 1987, pp. 98-112.

[Peyton87b] Peyton-Jones, S.L. *The Implementation of Functional Programming Languages,* Prentice/Hall International 1987.

[Peyton88] Peyton-Jones, S.L. "FLIC – A Functional Language Intermediate Code." In *SIGPLAN Notices,* August 1988, pp. 30-48.

[Peyton89] Peyton-Jones, S.L. and Salkild, J. "The Spineless Tagless G-Machine." In *Functional Programming Languages and Computer Architecture,* ACM, September 1989, pp. 184-201.

[Polych86] Polychronopoulos, C.D., Kuck, D.J., and Padua, D.A. "Execution of Parallel Loops on Parallel Processor Systems." In *International Conference on Parallel Processing,* 1986, pp. 519-527.

[Potter85] *The Massively Parallel Processor,* MIT Press 1985.

[Quinto84] Quinton, P. "Automatic Synthesis of Systolic Arrays from Uniform Recurrent Equations ." In *International Symposium on Computer Architectures,* 1984, pp. 208-214.

[Reddaw89] Reddaway, S.F. "Achieving High Performance Applications on the DAP." In *CONPAR 88,* Jesshope, C.R. and Reinartz, K.D. (Eds.), British Computer Society, Cambridge University Press, 1989, pp. 264-273.

[Rice89] Rice, M.D., Seidman, S.B., and Wang, P.Y. "A High-Level Language for SIMD Computation." In *CONPAR 88,* Jesshope, C.R. and Reinartz, K.D. (Eds.), British Computer Society, Cambridge University Press, 1989, pp. 384-391.

[Sabot88] Sabot, G. *The Paralation Model,* MIT Press 1988.

[Sharp91] Sharp, D. and Cripps, M. "Parallel Algorithms that Solve Problems by Communication." Tech. Rept. DoC 91/23, Department of Computing, Imperial College, 180 Queens Gate, London SW7 2BZ, U.K., 1991.

[Skilli90] Skillicorn, D.B. "Architecture-Independent Parallel Computation." *Computer* **23**, 12, December 1990, pp. 38-50.

[Spivey89] Spivey, M. "A Categorical Approach to the Theory of Lists." In *Mathematics of Program Construction,* van de Snepscheut, J.L.A. (Ed.), Springer-Verlag, 1989, pp. 399-408.

[Thinki87] *Connection Machine Model CM-2 Technical Summary,* Thinking Machine Corp. Technical Report HA87-4, April, 1987.

[Tombou89] Tomboulian, S. "Indirect Addressing and Load Balancing for Faster Solutions to Mandelbrot Set on SIMD Architectures." Tech. Rept. AN101, Application Note, MasPar Computer Corporation, 749 North Mary Avenue, Sunnyvale, CA 94086, July, 1989.

[Valian89] Valiant, L. "Bulk-Synchronous Parallel Computers." In *Parallel Processing and Artificial Intelligence,* Reeve, M. and Zenith, S.E. (Eds.), 1989, pp. 15-22.

[Wadler89] Wadler, P. "Linear Types can Change the World." Tech. Rept. University of Glasgow, Dept. of Computer Science, Lillybank Gardens, Glasgow, G12 8QQ, U.K., September, 1989.

[Walins90] Walinsky, C. and Banerjee, D. "A Functional Programming Language Compiler for Massively Parallel Computers." In *Lisp and Functional Programming,* ACM, June 1990, pp. 131-138.

[Watson89] Watson, I., Sargeant, J., Watson, P., and Woods, V. "The Flagship Parallel Machine." In *CONPAR 88,* Jesshope, C.R. and Reinartz, K.D. (Eds.), British Computer Society, Cambridge University Press, 1989, pp. 125-133.

[Wei-Ng90] Wei-Ngan, C. *Automatic Methods for Program Transformation,* Ph.D. dissertation, Imperial College, 180 Queens Gate, London SW7 2BZ, U.K., May 1990.