COMPILER CONSIDERATIONS AND RUN-TIME STORAGE MANAGEMENT FOR A FUNCTIONAL PROGRAMMING SYSTEM

Jose Nagib Cotrim Arabe

August 1986
CSD-860041

UNIVERSITY OF CALIFORNIA
Los Angeles

Compiler Considerations and Run-Time Storage Management for a Functional Programming System

A dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy
in Computer Science

by

José Nagib Cotrim Arabe

1986

© Copyright by
José Nagib Cotrim Arabe
1986

In memoriam of Nagib Arabe, my father

To Yvonette Cotrim Arabe, my mother

To Heloiza Emilia Blanc, *minha companheira*

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I wish to express my appreciation to my doctoral committee consisting of Professors Miloš D. Ercegovac (Chairman), Tomás Lang, Dan Berry, Bruce Rothschild and Richard L. Baker. I also wish to thank Professor Loyce Adams, who served as a member of the committee at an earlier stage. I am specially grateful to my advisor, Professor Miloš D. Ercegovac, who introduced me in the area of Functional Programming and suggested the problem. His illuminated guidance and our helpful discussions provided the necessary encouragement and confidence throughout this research.

Financial support to this work was provided by Coordenação de Aperfeiçoamento do Ensino Superior (CAPES), Ministry of Education, Brazil, through fellowship 3906/81; by ONR Contract N00014-83-K-0493, "Specification and Design Methodologies for High-Speed Fault Tolerant Array Algorithms and Structures for VLSI," and the State of California MICRO-Rockwell Grant "A High-Level Language Approach to Custom Chip Layout Design." Additional support was provided by Universidade Federal de Minas Gerais, Belo Horizonte, Brazil, which also granted me a leave of absence thus making possible this achievement. To all these entities I express grateful appreciation.

During my stay in Los Angeles, I have enjoyed the friendship of several people. My office mates T.M. Ravi, Paul Tu, Socrates Dimitriadis, Shun Cheung, Pak Chan, Martine Schlag and Miquel Huguet provided a nice working environment. My brazilian friends Valmir C. Barbosa, José Diaulas Palazzo Rolim and Frank Schaffa provided unforgetable moments of moral

support during lunch and coffee breaks.

My mother, Yvonette Cotrim Arabe, and my friend, Professor Christiano Gonçalves Becker, took care of my Brazilian affairs while I was absent; their hard work there made my life here easier.

Lastly, and most important, I must remember Heloiza Emilia Blanc, my wife, companion and friend, who sacrificed four years of her professional career to endure by my side the hardships of life abroad. Without her support and love this dissertation would not exist.

VITA

January 15, 1955	Born, Belo Horizonte, Minas Gerais, Brazil
1973-1977	Electrical Engineer, Universidade Federal de Minas Gerais, Brazil
1979-1982	Master of Science in Computer Science, Universidade Federal de Minas Gerais, Brazil
1977-1979	Systems Analyst, Computer Center, Universidade Federal de Minas Gerais, Brazil
1979-	Assistant Professor of Computer Science, Universidade Federal de Minas Gerais, Brazil
1984-1986	Post-Graduate Research Engineer, University of California, Los Angeles

ABSTRACT OF THE DISSERTATION

Compiler Considerations and Run-Time Storage Management
for a Functional Programming System

by

José Nagib Cotrim Arabe
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 1986
Professor Milos D. Ercegovac, Chair

This dissertation deals with the efficient execution of functional programs. First, we identify sources of inefficiency in the functional language FP that contribute to slowdown program execution irrespective of machine implementation. Then, a system for symbolic structural evaluation of FP programs is defined. The system is given a description of the structure of the input object and the FP program; based on this information and on basic algebraic relations, the system derives the structure of the result object without the need for the actual input object.

This system lays the groundwork for the implementation of a compiler that solves an FP program structurally in order to generate efficient run-time environment for the actual execution of the FP program. Algebraic equations are used to represent the structure and the location of FP objects in a given memory organization. The manipulation of these algebraic equations allows the structural solution of FP primitives at compile time; this process reduces

the amount of data replication and data movement required by the original FP program. The specific result is a more efficient run-time environment with respect to a given architecture for FP programs.

The approach is demonstrated by a comparison between the memory requirements of the compiler and of the conventional mode of implementation for FP, namely, interpretation. We show that the compilation approach indeed results in less data replication and less data movement.

Lastly, compilation issues for pipelined and multiprocessor systems are discussed. We show that the ideas implemented in the compiler are also useful when the machine model is not an uniprocessor. A number of common vector operations as well as some actual problems are implemented in FP and their execution performance is analyzed. Then, we discuss the use of compilation in the control of interconnection networks, a crucial element for multiprocessor architectures. Finally, a string-reduction architecture developed for the specific purpose of executing FP programs is examined; we show that this architecture also can take advantage of the compilation techniques.

CHAPTER 1
INTRODUCTION

This work describes a new approach to improve the efficiency in the execution of functional (or applicative) programs. The functional style of programming has been recently advocated as a paradigm for the development of software because of its mathematical basis, semantic elegance, ease of expressing implicit and explicit parallelism, expandability, and modularity.

We begin by identifying sources of inefficiency in the functional language FP [Back78] that are likely to slow down program execution irrespective of machine implementation. These sources of inefficiency are then considered as targets for optimization by the FP system. Basically, the problem can be stated as: "given an FP program and an architecture, generate an optimized run-time code." In other words, we are looking for FP *code optimization*, a problem well-known also for conventional languages. However, the complexity of code optimization in FP turns out to be smaller than the complexity of code optimization for conventional languages. This is due to the cleaner semantics of FP, with no side-effects and variables, factors that are a source of problems in the optimization of procedural languages. Also, FP has a powerful algebra of programs that can be used in the process of program transformation.

Then we describe the new approach to deal with those efficiency problems. We advocate the use of *compilation* techniques for FP programs to generate more efficient FP code. Algebraic equations are used to represent the structure and the location of FP objects in a given memory organization. The manipulation of these algebraic equations allows the solution of some FP primitives at compile time; this process reduces the amount of data replication and data movement required by the original FP program. The specific result is a more efficient run-time environment with respect to a given architecture for FP programs. This chapter is dedicated to the presentation of the motivations for the research and to a more precise definition of the problem.

1.1 Background

In recent years very active research has been done in the area of concurrent computation. This trend is justified by a number of facts:

a. many computations have a highly parallel structure;

b. VLSI technologies allow processors to be combined into large parallel structures;

c. greater demands for faster and more powerful computing resources.

This research into parallel computation has brought with it new problems in the creation of algorithms and in computer architecture design. One of the aspects of these new problems is the development of high level languages (HLLs) to enhance the programmability of highly concurrent systems.

HLLs have been developed to allow algorithmic specifications in a concise and machine-independent form. However, the most popular HLLs (such as FORTRAN, Pascal and Algol) were developed some time ago and are based on the so-called von Neumann architecture concept; these machines were designed to perform sequential operations on individual items of scalar data. As a result, conventional programming languages — also called procedural or imperative languages — enforce an artificial sequentiality in the specification of algorithms. This sequentiality not only adds verbosity to the algorithm, but may prevent an efficient execution of the algorithm on concurrent architectures. Various proposals have been made to attack such inadequacies. In the next paragraphs, some of them are outlined.

One approach to overcome these problems is the development of *vectorizing compilers* which try to recover parallelism from a sequential specification. This process is rather artificial since the user specifies a potentially parallel algorithm using a sequential language and then the compiler tries to infer the lost parallelism. It seems clear that to translate a parallel process to a sequential one is easier than to do the inverse process. This is because in the first case an arbitrary ordering suffices, whereas in the second case the process requires an analysis. Another serious limitation to this approach is the existence of variables and their associated side-effects on conventional languages. Lastly, algorithms coded cleverly to save memory or run faster on a sequential machine can prevent the compiler to find parallelism in the code. Nonetheless, some very good results have been obtained as can be seen in [Kuck81, Padu80, Arno82].

To increase the efficiency of this process, the user either has to be aware of the machine instruction set or of the method of detection of parallelism used by the compiler. Furthermore, the organization of the transfer of data to and from the memory, through an interconnection network, can require the use of low-level primitives and can critically affect the performance of a program. The only justification for this approach is to avoid the rewriting of software, given that software is more expensive than hardware. Therefore it should be considered only as a temporary approach for the move from sequential machines to parallel machines.

Another approach is to extend conventional programming languages, in particular FORTRAN, by incorporating *parallel constructs*. This allows the programmer to explicitly exhibit the parallelism in the problem. It has been a widely used approach, as can be seen by languages such as CFD (a FORTRAN-like language for the Illiac-IV) [Stev75], BSP FORTRAN [Burr77], and Vectran [Paul75]. Some shortcomings can be pointed out:

a. since the languages are based in old concepts, they inherit their deficiencies, most important of them the presence of variables and side-effects;

b. although recent efforts have been made towards standardization, these languages are generally modified versions of old languages to suit a given architecture; they incorporate details of that machine architecture (in fact, most of the parallel constructs mimic the assembler language parallel constructs of the machine), making the languages non-portable;

c. even languages that are claimed to be architecture-independent [Perr79] suffer from the fact that they are based on the von Neumann model of computation which make them unsuitable to novel massively parallel architectures.

The class of multitasking languages such as Ada and Modula-2 also present problems, since they require explicit programmer concern with the creation and synchronization of multiple tasks. This clearly adds complexity and cost to the software development process. Furthermore, within a specific process, these languages have the same lack of parallelism of the earlier conventional languages.

One exception among the old languages is APL [Iver62]. APL is based on mathematical concepts and notations, and allows a concise and elegant expression of problems. By having arrays as basic data types and powerful operators to manipulate them, APL can express parallelism consistently. But, as noted by [Back78], APL still has variables and side-effects, and presents some of the shortcomings of von Neumann languages.

In recent years, various scientists have advocated the use of a new class of programming languages to overcome some of the deficiencies of conventional languages. The class of *applicative* or *functional* programming languages has been developed with the aim of having the following characteristics fulfilled: they should be as machine-independent, natural and high-level as possible; they should enable the essence of an algorithm to be captured in a program, by allowing the elimination of any arbitrary detail not directly associated with the problem itself; and they should be amenable to

correctness proofs with a minimum of analysis.

Various proposed languages can be classified as functional languages, among them: Backus' FP [Back78], SISAL [McGr83], Hope [Burs80], Val [McGr79], Id [Arvi78], and KRC [Turn82].

The important characteristics of functional languages are:

a. the program is a function in the mathematical sense;

b. the basic operation is function application: the program is applied to the input and the resulting value is the program's output;

c. the value of an expression depends only on its textual context, not on computational history (no concept of present state, program counter or storage as in von Neumann machines); this property is called *referential transparency* [Back72];

d. absence of variables and side-effects (although some of the languages have an assignment statement – for example Val, Id and SISAL are called single-assignment languages – it is simply a notational convenience for binding an expression to an identifier).

Imperative languages promote a style of programming based on naming of elementary cells, assignments to these cells, and repetition of elementary actions. The functional programming style, on the other hand, does not depend on these three actions. The simple and uniform data objects (e.g., lists) allow the design of data structures without concern for memory cells; instead of being assigned, values are produced by function application and passed on

to other functions; and functional forms and operations that distribute over the data objects reduce the reliance on repetition. Therefore, one advantage of functional programming is that it allows programs to be written at a higher level than imperative programming. Other advantages claimed by functional language proponents are:

a. functional languages have concise and simple semantics;

b. functional languages are expandable and modular;

c. compact notation allows more algorithm to be expressed per line of code. Evidence suggests that number of lines of correct code per day is roughly constant for a given programmer, independent of the language used [Wass82]. Programming experience seems to indicate that functional languages do in fact increase programmer productivity. Some users and designers of functional languages have made claims of improved productivity [Burs80, Turn81, Morr80];

d. freedom from side-effects;

e. functional programs are easier to verify because proofs can be based on the concept of a function rather on the more cumbersome notion of state transitions [Back78];

f. functional programs can represent implicit and explicit parallelism.

Of course, these advantages have their price in terms of execution efficiency. The inefficiency comes mainly from the fact that many objects are created and discarded dynamically. The dynamic creation of objects such as

lists and arrays causes excessive data movement and data replication during the execution of functional programs. These issues will be discussed in detail in Section 1.3.

1.2 The FP Functional Language

We choose to use in this work the functional programming language FP because it possesses excellent formal properties, among them referential transparency, and it has a powerful algebra of programs [Back78]. Here we briefly introduce FP. It is basically the same as Backus' FP except that it has a few more primitive functions. The complete description of the syntax and semantics of the language is given in Appendix 1; here we present only the most significant features.

The FP language comprises (1) a set of *data objects*, (2) a set of *primitive functions*, (3) an operation, *application*, (4) a set of *functional forms*, and (5) a set of *definitions*.

Data Objects

The set of data objects consists of atoms and sequences. Numbers, characters and boolean values (**T** and **F**) are all *atoms*. A sequence with n elements $x_1, x_2, \cdots, x_n$ is denoted by $<x_1, x_2, \cdots, x_n>$. Each element x_i may be any data object. The *empty sequence* ($n = 0$) is denoted by <>. A special atom, *bottom*, represented by **?**, denotes the value returned as the result of an undefined function application.

Primitive Functions

Below is an informal definition of some of the primitive functions of FP; the complete definitions can be found in Appendix 1. Note that *function application* is denoted by ":", as in $f{:}x$.

Structure manipulating functions:

Selectors: **1**: $\langle x_1, x_2, \cdots, x_n\rangle = x_1 \ \ (n\geq 1)$,

2: $\langle x_1, x_2, \cdots, x_n\rangle = x_2 \ \ (n\geq 2), \cdots$

tail: $\langle x_1, x_2, \cdots, x_n\rangle = \langle x_2, \cdots, x_n\rangle \ \ (n\geq 1)$

transpose:

$\langle\langle x_{11}, x_{12}, \cdots, x_{1m}\rangle, \cdots, \langle x_{n1}, x_{n2}, \cdots, x_{nm}\rangle\rangle$

$= \langle\langle x_{11}, x_{21}, \cdots, x_{n1}\rangle, \cdots, \langle x_{1m}, x_{2m}, \cdots, x_{nm}\rangle\rangle$

distribute-right: $\langle\langle x_1, x_2, \cdots, x_n\rangle, y\rangle$

$= \langle\langle x_1, y\rangle, \langle x_2, y\rangle, \cdots, \langle x_n, y\rangle\rangle$

distribute-left: $\langle y, \langle x_1, x_2, \cdots, x_n\rangle\rangle$

$= \langle\langle y, x_1\rangle, \langle y, x_2\rangle, \cdots, \langle y, x_n\rangle\rangle$

append-left: $\langle y, \langle x_1, x_2, \cdots, x_n\rangle\rangle = \langle y, x_1, x_2, \cdots, x_n\rangle$

etc.

Arithmetic and boolean functions:

+: $\langle x,y\rangle = x+y$, −: $\langle x,y\rangle = x-y$, etc. (*, /, **and**, **or**, **not**)

Predicates:

null: x = **T** if x is the null sequence, **F** otherwise

eq: $\langle x,y\rangle$ = **T** if x is identical to y, **F** otherwise

etc.

Identity function: id: $x = x$

Functional Forms

Functional forms are used to create new functions by combining existing functions:

composition: f @ g: $x = f : (g : x)$

construction: $[f_1, f_2, \cdots, f_n]$: $x = \langle f_1 : x, f_2 : x, \cdots, f_n : x \rangle$

constant: $\%x$: $y = x$, for any objects x, y

conditional: $(f \rightarrow g\ ;\ h)$: $x =$ if $f : x$ then $g : x$ else $h : x$

apply-to-all: $\&f : \langle x_1, x_2, \cdots, x_n \rangle = \langle f : x_1, f : x_2, \cdots, f : x_n \rangle$

right-insert: $!f : \langle x_1, x_2, \cdots, x_n \rangle = f : \langle x_1, !f : \langle x_2, \cdots, x_n \rangle \rangle$

where $!f : \langle x \rangle = x$ and $!f : \langle \rangle = e_f$

for some specific value e_f in the range of f

Definitions

A definition has the form {*name form*} where *name* is the name of the function being defined and *form* is a functional expression made of primitive functions, function names (other definitions) and functional forms. As an example, the following is the recursive definition of the factorial function:

```
{factorial    (eq0 -> %1;  * @ [id, factorial @ sub1] ) }

{eq0          eq @ [id, %0] }

{sub1         - @ [id, %1] }
```

Execution Models for FP Programs

The FP language may be characterized as a *reduction language* since it supports a reduction-style program execution. In contrast with procedural languages, which are built of and executed as sequences of simple operations called instructions, reduction programs are built from nested expressions. In reduction language programs, the equivalent to an instruction is the *application* of a function to an object, the result being another object. A reduction language program is an expression equivalent to the result of its evaluation in the same sense that (2+2) is equivalent to 4.

Two basic approaches have been identified [Trel82] toward executing reduction programs: *string reduction* and *graph reduction.* The basis of string reduction is that a program is manipulated in place: each application of a function to an argument is textually replaced by an equivalent expression, and expressions are not shared. Commonly used sub-expressions must be replicated throughout the program where necessary. In graph reduction, implicitly shared references (pointers) to expressions are manipulated.

For illustration purposes, consider the following FP definition of the arithmetic expression (b+1) * (b-c):

```
{a    * @ [t1, t2] }
{t1   + @ [b, %1] }          {t2   - @ [b, c] }
{b    %4 }                   {c    %2}
```

Figure 1.1 shows how the expression is evaluated using string reduction. Note that the reference to a is textually substituted by its definition.

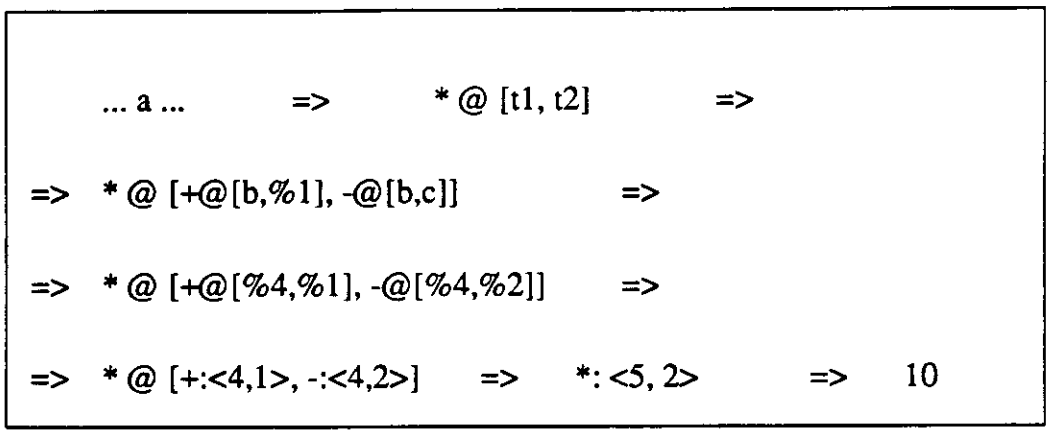


Figure 1.1 - String Reduction of (b+1) * (b-c) **[Trel82]**

Then, possibly in parallel, the definitions of **t1** and **t2** replace their occurrences in the expression. Note also that each reference to **b** is followed by a substitution by the value of **b**, that is, no sharing of the value of **b** occurs. At later stages, the reducible sub-expressions +: <4,1> and -:<4,2> are replaced by their result and, finally, the multiplication is executed leaving the final answer in place of the whole original expression. This process happens for each occurrence of **a** in the body of the FP program.

Figure 1.2 illustrates the graph reduction for the same program. When the reference to **a** is found, instead of a copy of the definition, the reference is traversed in order to reduce the definition and return the actual value. One way of identifying the original source of the demand for **a**, and thus support the return, is to reverse the arcs by inserting a source reference in the definition. The traversal of the definition and reversal of the references continues until constant arguments, such as **b** and **c**, are encountered. Reduction of the sub-expressions starts with the rewriting of the addition and subtraction an shown

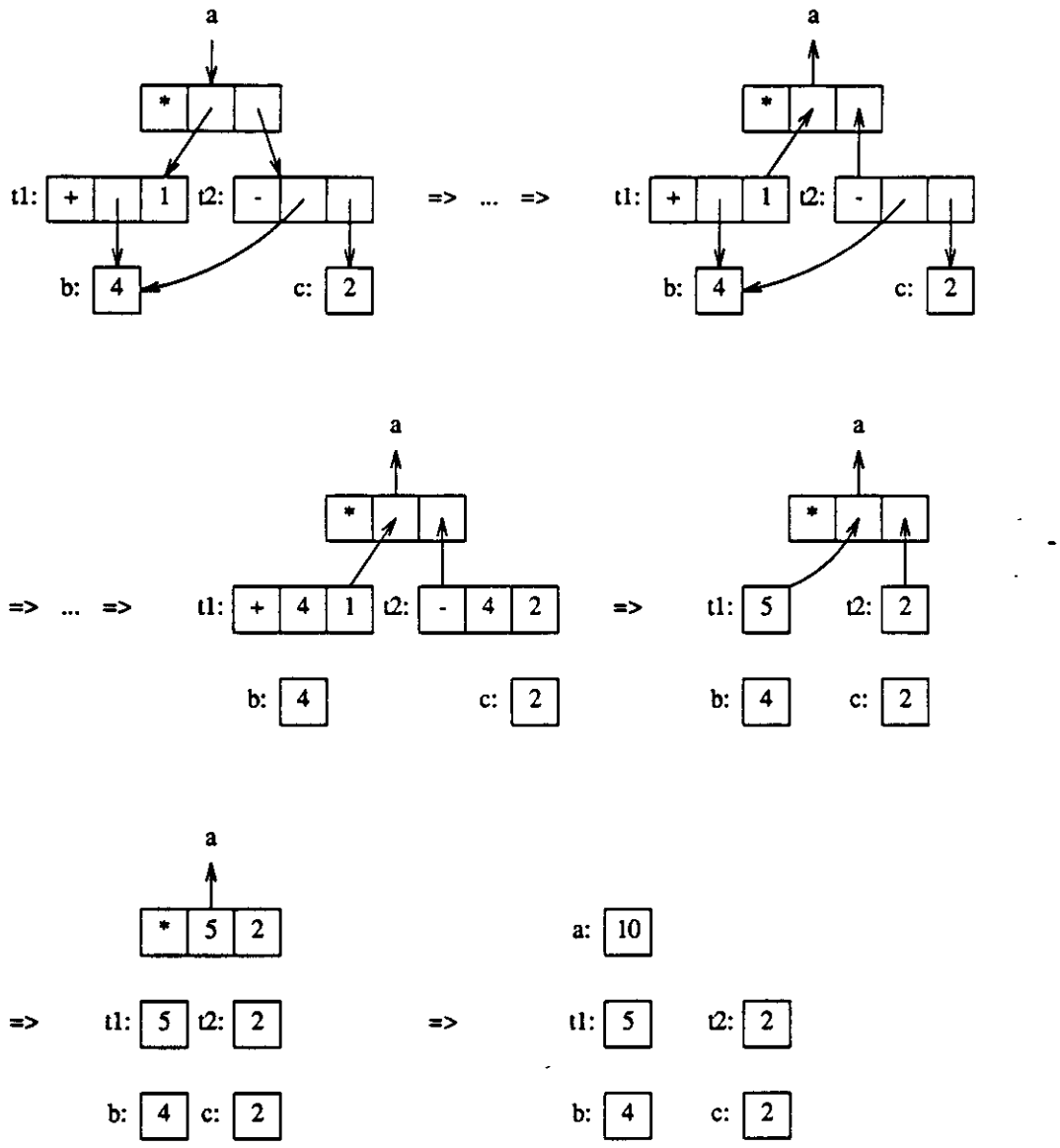


Figure 1.2 - Graph Reduction of (b+1) * (b-c) **[Trel82]**

in Figure 1.2. The process continues until the value of **a** is calculated and returned to the original demand. Any further reference to **a** immediately receives the value 10; no recalculation is done, as is the case in string reduction. Note that if there are no further references to **b, c, t1** and **t2** they can be garbage collected.

Another issue when executing reduction languages concerns the choice of the next application to be reduced. There are two possible rules: in an innermost selection rule, the applications selected are the most deeply nested; in an outermost selection rule, the un-nested applications are those selected.

All outermost reduction architectures are *demand-driven* [Turn79, Clar80, Kell79, Darl81], a term used to define computer organizations where instructions are only selected when the value they produce is needed by another, already selected instruction. This means that an instruction is executed only when its result is demanded by some other instruction. Outermost selection rules are best suited for a graph reduction model of execution. This is because, in the case of using innermost rules for graph reduction, no real use is made of the graph reduction's by-reference data mechanism: all subexpressions would be reduced before the functions referring to them (the only references in functions would be to values) and there would be no sharing of subexpressions.

On the other hand, innermost reduction architectures are *data-driven* [Arvi82, Gurd85, Mago80, Berk75], since an instruction is executed only when all its arguments are available (evaluated). This means that no coercions (demands) take place and that all functions have their arguments evaluated

whether this is necessary or not, as occurs in the *data-flow* model of computation [Denn80]. Innermost selection rules are best suited for a string reduction model of execution. This is because string reduction uses a by-value data mechanism, where copies of actual arguments are generated for each formal parameter occurrence. Figure 1.3 summarizes the above discussion.

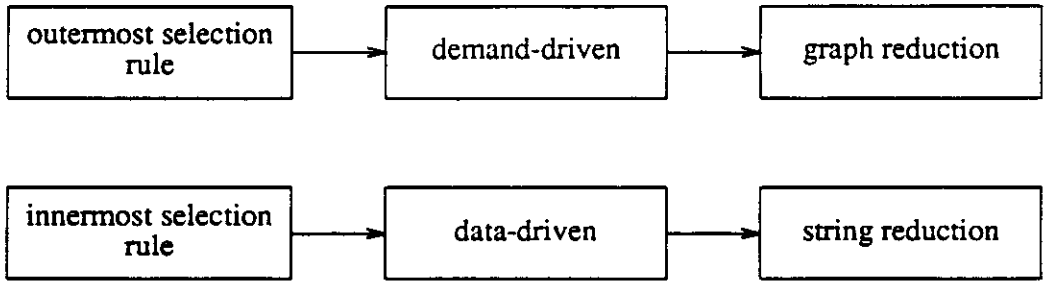


Figure 1.3 - Execution Models for FP

1.3 Description of the Problem

In the Section 1.1 we argued that functional languages offer advantages with respect to ease of problem expression and also offer a great potential for the exploitation of implicit and explicit parallelism. However, they are not widely used in the real word of programming. What are the reasons?

One aspect of the problem is resistance to change. Programmers have to be reeducated to think in functional terms and not in procedural terms; this can be a long and arduous process. Also, economic reality dictates that the whole body of already built imperative software must be used.

Another aspect is that functional languages have a reputation of being very inefficient to execute. It is clear that the user community must be convinced not only that functional languages are appropriate tools, but also that they can be efficiently executed. Therefore, this work will deal with the problem of efficiently executing functional programs.

Several reasons can be listed for the performance degradation in functional programs:

a. von Neumann architectures are not suited to FP execution, mainly because FP does not reflect the structure and operations on those architectures. That is, FP is not based on naming of memory cells (variables), assignment to these cells, and repetition of elementary actions;

b. programs are based in the manipulation of lists, which require a general list manipulation system with garbage collection, causing an associated overhead;

c. lack of destructive updating; a new copy of a structure is (logically) needed when the structure is modified;

d. in general systems are interpretative rather than compiled. The reason for this is that programming languages that adopt dynamic binding between objects and types, which is the case of FP, are processed more naturally by interpretation. In these languages, there generally is not enough information before run-time to generate code for the evaluation of expressions involving objects of unknown type. This makes languages with dynamic type binding interpretation-oriented, whereas

languages with static binding are translation-oriented. However, dynamic binding does not completely preclude the use of compilation, which oftentimes removes sources of inefficiency.

Even novel architectures that have been proposed to directly execute functional languages have not been very impressive in terms of performance. Some of these new architectures are:

a. *data-flow machines:* MIT Data Flow Project [Denn80], Arvind's U-Interpreter [Arvi82], Rumbaugh's Data Flow Multiprocessor [Rumb77], Manchester Data Flow Machine [Gurd85], and LAU [Syre77].

b. *string-reduction machines:* Magó's Tree Machine [Mago80], Berkling's Reduction Machine [Berk75], and Kellman's Machine [Kell83].

c. *graph-reduction machines:* Turner's Combinator Reduction Machine [Turn79], SKIM [Clar80], AMPS [Kell79], and ALICE [Darl81].

Since most of the proposed architectures are "paper-designs," real performance evaluations are not available. However, performance predictions by analysis and simulation corroborate the fact that many issues remain to be solved with respect to efficiency. Some of the new machines have actually been built; again, performance results show that they still fall behind current conventional high-speed machines [Gurd85].

It has been recognized by the designers of these machines that one of the critical points is memory allocation and management. Most machines handle very poorly regular data structures such as arrays and vectors. This make them non-competitive with existent super-computers and easy targets to criticism. Furthermore, there is a big class of problems that manipulate arrays and it seems unlikely that functional and data-flow architectures will succeed if they do not handle well this class of problems.

The easiest way to make data structures free of side-effects in functional languages is to forbid sharing or overlapping of data. However, this can be prohibitively expensive since it requires each structure to be completely copied whenever its value is modified, even if only one element is changed. To circumvent this problem, Dennis [Denn74] proposed to store arrays as trees, with array elements at the leaves, allowing most access and manipulation operations to be performed in logarithmic time. In Dennis' proposal, structures are shared whenever possible by using reference counts. Each node of a structure has a reference count, which is the total number of pointers to that node from other nodes. Then, if a copy *B* of array *A* is created, the pointer for *B* points to the same root as the pointer for *A*. This approach has the disadvantage that the complexity of many simple access and data manipulation operations is increased.

Another proposal to avoid excessive storage demand and slow access time for regular data structures in functional languages is the so-called *I-structure*, proposed by Arvind and Thomas [Arvi80]. These are array-like data structures whose storage is allocated before expressions to produce them

are invoked. To improve parallelism, it is possible for a part of a program to attempt to read an element before the creation of that element. Therefore, a presence bit is associated with every element, and an attempt to read an empty location defers the read operation. Checking for those deferred reads on every write is the main cause for performance degradation in I-structures. Another problem is to optimally distribute the I-structures over many processors to minimize traffic through the network that interconnects the processors.

While the approaches described above are different implementation attempts to overcome the inefficiencies in the representation and manipulation of regular data structures in functional languages, there are other sources of inefficiency that, without regard to implementation, will be likely to contribute to slow down program execution. They are listed below.

Excessive data movement: While nobody expects a programmer to write code such as: **trans @ trans**, or **reverse @ reverse**, or even **1 @ reverse** (which is the same thing as **last**), these cases can occur in a subreptitious manner. Suppose a function **f1** does a given job and finishes it by reversing the result list. Suppose also that another function needs exactly the result of **f1** but without the reversing step. Naturally, a programmer can take advantage of the existence of **f1** and write **reverse @ f1**. At execution time, the undesired encounter **reverse @ reverse** will occur; a system that blindly executes such segment of program will spend some, maybe long, time doing operations with a null effect.

Note also that influence of programming style can generate excessive data movement and therefore bring inefficiencies. Suppose a programmer writes this piece of code in FP: **1 @ trans**. If this code is to be applied to a matrix, the intention is to have as a result the first column of the matrix. It is clear that the same objective can be achieved with the following function: **&1**. If the FP code is directly interpreted by a machine, in the first program the matrix is first transposed, while in the second no such data movement occurs, there being only selection operations. Therefore, the first version is likely to be slower no matter how the system is implemented.

To avoid this type of problem, either the programmer has to be aware of the potential inefficiency of the first version or the system has to be smart enough to avoid the actual transposition of the matrix. It is important to note here that both versions of the program are very clear in their intent, i.e., to select the first column of the matrix. Therefore, we do not advocate that the second version is clearer than the first one; such a conclusion is at least debatable. In conclusion, if the programmer has to deal with efficiency questions of this nature, one of the very first motivations to use functional languages, i.e., to be machine-independent, high-level and as natural as possible, will be no longer valid. It remains the option of building a system that detects such sources of problems; some approaches will be discussed in the next section.

The issue of data movement is a serious one in the functional programming style. Because there are no variables in FP, a function locates its arguments by their position within the input object; thus operations that direct data movement (transposition, selections, reversings) occur frequently in func-

tional programs. Therefore, one of the objectives of any implementation of a functional programming system should be the minimization of data movement.

Redundant computations: Another potential source of inefficiency in functional programs is that sometimes a straightforward execution causes the same computation to be performed repeatedly. For example, with unlimited resources, the execution of [f1@reverse, f2@reverse] can take the same time as the equivalent program [f1, f2] @ reverse. However, in the real situation of limited resources, it is quite recommended that the system discovers the common subexpression in the first case to avoid its recomputation. The issue is a delicate one, because sometimes it is difficult to detect common subcomputations that would reduce execution time; and sometimes, with enough resources, it may be desirable to ignore the redundancy.

Excessive data copying: It is well known that the introduction of redundancy often brings opportunities for parallelism. For example, the expression $a(bcd+e)$ can be executed in four steps with only one functional unit. On the other hand, its equivalent $abcd+ae$, obtained by applying the arithmetic law for the distribution of multiplication over addition, can be executed in three steps using two functional units. Note that distribution has introduced one extra operation and that two copies of a are needed. It is this redundancy that enables the speedup gain.

However, data replication does not always lead to gains in speedup. Suppose that the piece of code [1, trans@2] is to be applied to object <A B>, where A and B are matrices. If this code is executed according to a string

reduction semantics, for example, as in Magó's Machine [Mago80], the following steps are obeyed:

1. <1: <A B>, **trans@2**: <A B>>

2. <A, **trans**: B>

3. <A, B'>, where B' is B transposed.

It is clear that unnecessary replication of data occurs at step 1. The semantics of this piece of code is only to transpose the second argument, leaving the first as it is. While implementations such as graph reduction machines, which use pointers to the real data, do not have this as a critical problem, string reduction machines such as Magó's machine can have performance degradation because of excessive data replication. Therefore, any implementation of a functional programming system should minimize or even eliminate unnecessary data replication.

The issue is partially treated by Magó in [Mago81]. He proposes a temporary overriding of the string representation used in the machine to avoid copying sufficiently large operands. There are two main problems in the proposal: first, it is not clear even for his machine what is meant by a sufficiently large operand; second, the approach can introduce sequentiality in the execution of the construction functional form, an otherwise parallel specification tool in FP. Furthermore, for situations as the one described above, the replication of data in step 1 is unproductive no matter the sizes of A and B. Realizing that, Magó proposes the introduction of a new functional form "apply to the rightmost operand" – **AR** [Mago84]. In this case, the application above

would be written **AR trans:** <A B> and indeed no excessive data copying would occur. The clear shortcoming of this proposal is that we cannot keep incorporating *ad hoc* primitives into the language to suit the occasion.

The problem of data copying is more critical when programming with regular data structures such as vectors and matrices. In functional programming, to modify a regular structure means to modify a *copy* of the whole structure, even if only a small part of the structure is to be modified. Clearly, the expense of copying large structures cannot be ignored – indeed one might try to *limit parallelism* in order to avoid copying [Mago84]. There are a number of ways to avoid this problem. The most brute-force is to allow some impure operators with side-effects (like RPLACA and RPLACD in Lisp). Clearly this is a non-solution, since it destroys referential transparency which was claimed to be one of the chief advantages of functional languages.

Another approach to attack this problem is described in [Huda85]. The authors describe a combination of static compilation techniques and dynamic run-time techniques to avoid excessive copying of arrays. Statically, if it can be determined at the moment an array is to be updated that no other function depends on that array, it is modified in place. If this analysis fails, they propose limiting the parallelism if the objective is to avoid copying at all costs or to use a modified reference counting scheme that determines dynamically if copying can be avoided.

In summary, the problem for which this work will propose some solutions is the excessive data movement and data replication that occurs in functional languages as a consequence of the functional semantics. This problem is

more serious when the data structures involved are of regular nature such as vectors and matrices and we show later how compilation techniques can take advantage of regularity in lists and treat them as arrays in order to improve performance of FP programs at run-time.

The next section examines some related work that has been done in the area of functional programming transformation with the objective of increased performance of execution.

1.4 Related Work

We examine two basic approaches, *schema methods* and *folding-unfolding methods*, that have been explored in the literature to remove inefficiency in the execution of functional programs. Figure 1.4 schematically represents both techniques.

The first approach applies source-to-source program transformations. Wadler [Wadl81] describes a set of functional forms that possess a complete set of four transformation rules that resembles algebraic manipulation. The objective is to convert applicative style programs to more efficient equivalents to be executed in a sequential machine. Basically, the transformations eliminate unnecessary list creations and traversals and remove some function call overhead. However, the author conceded in a later work [Wadl84] that the hope of extending the set of rules to handle additional functions proved to be difficult. He then developed a uniform procedure, which he calls a listless transformer, that performs lazy evaluation and garbage collection at compile time. The transformer uses symbolic evaluation to unfold function definitions

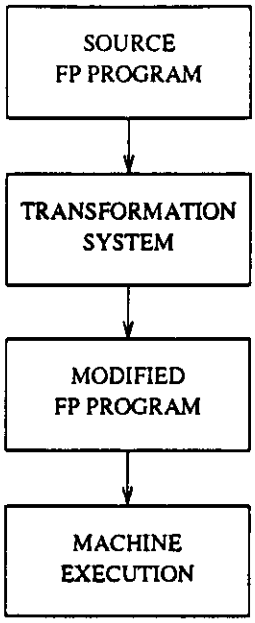


Figure 1.4 - Source-to-Source Program Transformation Approach for Optimization of FP Programs

and programs are transformed using an optimization technique called tail-recursion elimination. The net effect is the elimination of intermediate lists where possible.

In [Bell84], source-to-source transformations based on algebra of FP are applied to programs in order to minimize the number of intermediary sequences used in iterative programs. The objective is to optimize applicative expressions to run in a von-Neumann machine.

The implementation of a compiler for Lazy ML, a strongly-typed, statically scoped language with lazy evaluation is described in [Augu84]. Among the features of the compiler is the use of source-to-source transformations on the original program which is claimed to generate more efficient code. Other

works on the same line are [Isla81] and [Kieb81].

All the works described above, based on the so-called *schema method*, use a catalog of predefined proven transformations on various program schemes. They rely on pattern matching algorithms to discover the templates that can be transformed and lack some generality, since all transformations must be predefined. In these works, transformations are expressed as schemas containing patterns; a program must be recognized as being an instance of a pattern in the transformation schema. The effectiveness of this approach is always limited by the difficulty in recognizing instances of the schemas. Existing techniques are *ad hoc* and incomplete, and tend to rely on manual intervention. The paper [Givl84] examines the pattern-matching problem. By noting that the conditional functional form is impossible to handle in a manner that is complete and fast, the authors partition the matching problem in two theories: a conditional-free sub-theory and a sub-theory of conditionals.

A different approach, presented in [Mann75, Mann79, Burs77], is based on the so-called *folding-unfolding method.* An expression is unfolded by expanding a function call using the function's definition until an instance of the original expression is found. This is then folded (replaced by a function call). The main result of this method is to eliminate tail-recursion by transforming the function into iterative form. This technique clearly improves performance in a von Neumann machine; however it can reduce the potential for parallelism for certain functions. Although more general than the schema methods, it relies on heuristic search and is more complicated.

Finally, one approach that is more close to the one proposed in this work was first presented by Abrams in his APL machine [Abra70]. Abrams developed a technique called *beating*, in which certain data manipulation operations are implemented by performing transformations on array descriptors rather than on the arrays themselves. He also introduced a type of *lazy evaluation* for APL, which he called *dragging*, in which a compiler defers performing certain operations by transforming code which a low level interpreter must later be called upon to execute. His work was further extended by [Mint76, Budd84, Guib78]. Although Abrams's work was developed with a von Neumann machine as model (for sequential execution), several of his ideas can be well applied to a parallel framework. No similar work has been done for applicative languages.

1.5 Outline of the Dissertation

This chapter has presented the definition and scope of the problem we propose to treat in the remaining of this Dissertation. We briefly presented the characteristics and advantages of *functional languages*, focusing attention on sources of inefficiency in the execution of functional programs written in FP. These sources of inefficiency are then considered as targets for optimization by an FP system. We also discussed some work done by other researchers that is related with this research.

In Chapter 2 we define a system for symbolic structural evaluation of FP programs. The system is given a description of the structure of the input object and the FP program; based on this information and on some basic alge-

Finally, one approach that is more close to the one proposed in this work was first presented by Abrams in his APL machine [Abra70]. Abrams developed a technique called *beating*, in which certain data manipulation operations are implemented by performing transformations on array descriptors rather than on the arrays themselves. He also introduced a type of *lazy evaluation* for APL, which he called *dragging*, in which a compiler defers performing certain operations by transforming code which a low level interpreter must later be called upon to execute. His work was further extended by [Mint76, Budd84, Guib78]. Although Abrams's work was developed with a von Neumann machine as model (for sequential execution), several of his ideas can be well applied to a parallel framework. No similar work has been done for applicative languages.

1.5 Outline of the Dissertation

This chapter has presented the definition and scope of the problem we propose to treat in the remaining of this Dissertation. We briefly presented the characteristics and advantages of *functional languages*, focusing attention on sources of inefficiency in the execution of functional programs written in FP. These sources of inefficiency are then considered as targets for optimization by an FP system. We also discussed some work done by other researchers that is related with this research.

In Chapter 2 we define a system for symbolic structural evaluation of FP programs. The system is given a description of the structure of the input object and the FP program; based on this information and on some basic alge-

braic relations, the system is capable of deriving the structure of the result object without the need for the actual input objects. Since some restrictions on the structure of objects is imposed by the algebra, we make an analysis of a collection of real FP programs to ascertain that real gains can be obtained by the use of the algebra.

The algebra of Chapter 2 lays the groundwork for the implementation of a compiler that is described in Chapter 3. The compiler solves an FP program structurally in order to generate efficient run-time environment for the actual execution of the FP program. Algebraic equations are used to represent the structure and the location of FP objects (inputs) in a given memory organization. The manipulation of these algebraic equations allows the structural solution of FP primitives at compile time; this process minimizes the amount of data replication and data movement required by the original FP program.

Chapter 4 investigates the effects of compilation on a conventional uniprocessor model. We compare the approach described in Chapter 3 with the conventional mode of implementation for FP, namely, interpretation. We show that, for the class of programs characterized in Chapter 2, the compilation approach results in less data replication and less data movement.

Chapter 5 discusses compilation issues for pipelined and multiprocessor systems. We show that the ideas developed in the previous chapters are also useful when the machine model is not an uniprocessor. The first section deals with vector processing and pipelined computers. A number of common vector operations as well as some actual problems are implemented in FP and their execution performance is analyzed. Then, we discuss the use of

compilation in the control of interconnection networks, a crucial element for multiprocessor architectures. Finally, a string-reduction architecture developed for the specific purpose of executing FP programs is examined; we show that this model also can take advantage of the compilation techniques.

Finally, Chapter 6 presents some concluding remarks and suggestions for future research. In summary, the major contributions of this research are:

1. The development of an algebra of structural transformations for FP programs that is used as a basis for the implementation of compiler techniques for the FP system.

2. A compilation technique which minimizes the amount of data replication and data movement during the execution of FP programs. This technique uses algebraic equations to represent the structure and the location of FP objects (inputs) in a given memory organization. The manipulation of these algebraic equations allows the optimization of FP programs at compile time.

3. The identification of areas where the compilation of FP programs can enhance the execution performance for pipelined computers, interconnections networks and non-von Neumann architectures.

CHAPTER 2
SYMBOLIC STRUCTURAL EVALUATION FOR FP

The FP primitives can be divided in two main categories:

1. *computational primitives* that generate atoms based on the atoms of the input object; and

2. *structural primitives* that do not create new atoms; they merely manipulate the atoms within an object (e.g., **trans, reverse**), possibly leaving some out (e.g., selectors, **last, tl**) or replicating others (e.g., **distl, distr**).

Correspondingly, the cost of executing an FP program can be divided between computational costs and structural costs. Clearly, one way to reduce execution time of an FP program, as discussed in the previous chapter, is to reduce the data movement and data replication required by the algorithm. This can be achieved by gathering information on the structure of the algorithm and of the input object, and solving the structural primitives using a symbolic evaluator based on the algebra of FP.

In the introduction article to FP [Back78], Backus defines an associated algebra of FP programs. He demonstrates the power of the algebra by proving the correctness and equivalence of some FP programs. Others researchers also made some contributions on this algebra, such as [Will82]. Further use of the

algebra has been shown in Section 1.4, where some systems designed to improve the execution efficiency of FP programs make use of the rules of the algebra. Below, another use of the algebra is described and explored.

2.1 An Algebra of Structural Computations

We define a system for symbolic structural evaluation of FP programs as follows. The system is given a description of the structure of the input object and the FP program; based on this information and on the basic algebraic relations presented below, the system is capable of deriving the structure of the result object without the need for the actual input objects. In other words, this system defines an algebra of structural computations for FP programs. This algebra will be used as a basis for the implementation of the compiler, described later in this work, which solves an FP program structurally in order to generate efficient run-time environment for the actual execution of the FP program.

2.1.1 Symbolic Computations

An FP function f specifies how a data object d is mapped to another data object $f(d)$. If we consider only the structure of d and $f(d)$, f can be viewed as mapping the structure of d to the structure of $f(d)$, and we can associate with f a function f' which will define the mapping of the structures *only*. If we let D be the set of data objects and S be the set of structures of the elements of D, we can view the relation between f and f' according to the diagram of Figure 2.1.

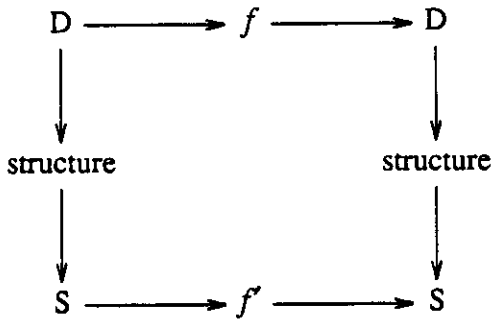


Figure 2.1 - Symbolic Structural Transformations

The function *f'* performs the same computation as *f*, except that it ignores the details irrelevant to structures. In this sense, we can view the computation performed by *f'* as a *symbolic structural evaluation.* By using symbolic evaluation, we can often deduce the structure of the result object without actually executing the function. For example, for the FP function + we know that the structure of the input object *must* be a sequence of two numbers and that the structure of the result is a number. We do not need to execute the program with real data to deduce this information.

However, two problems exist with this type of symbolic evaluation. First, a program that contains a conditional functional form as in the following example is not amenable to solution by symbolic evaluation:

```
{f   (= @ [1, 2] -> %<>; id)}
```

In this function the structure of the output object depends on the *values* of the input object; therefore the result of a symbolic evaluation system would be non-deterministic.

The second problem is that symbolic evaluation cannot capture the semantics of the *bottom* object as it is defined in conventional FP. This is because a computation in the structural domain will not have the information necessary to decide whether a computation terminates and produces proper values or not. For example, the division function expects a pair of numbers and is supposed to deliver a number as result. However, it can deliver *bottom* if the input object is not a pair of numbers or, even if it is, if the second number is zero. Clearly, a symbolic structural evaluation system cannot capture the behavior associated with this last case. On the other hand, there are instances where a structural evaluator can detect inconsistencies. For example, if an arithmetic function is applied to anything other than a list of two atomic elements, the result is undefined (e.g., +: <1 2 3>≡ ?). Therefore, we define a *structural bottom*, Λ, to capture such cases.

Although symbolic structural evaluation cannot always be performed, it is important to realize that a *partial* evaluation can be done on FP programs that present the obstacles described above. That is, the symbolic evaluator can solve the program structurally up to the point where a restriction is found, and then leave the remaining portion of the program to be solved when the data values are known.

2.1.2 Basic Relations for Primitive FP Functions

We begin by defining the *structure* of an FP object. This definition singles out atoms and finite sequences as the fundamental structures for FP objects.

Definition: The set S of *structures* is defined in the following way:

(1) atoms $\in S$;

(2) if $s_1, s_2, \cdots, s_n \in S$ then $<s_1, s_2, \cdots, s_n> \in S$;

(3) **length**: $< s_1, s_2, \cdots, s_n> \equiv n \in S$;

(4) FP-defined objects that are argument of the constant functional form also belong to S.

Only the above belongs to S.

We denote $\sigma\,(f:s)$ the structure of the object resultant from the application of f to an object of structure s.

Operationally, we represent the structure of FP object as follows:

a. Atoms (numbers, characters and boolean values) have structure a. Note that if we assign different types to these atoms (like *num* to numbers, *char* to characters, and *bool* to boolean values), we go into more detail than needed for a structural evaluation system. Clearly, the function + expects a lists of two numbers, and the function **and** expects a list of two booleans; however, from the structural point of view, both expect a list of two atoms. This is sufficient for the system we are describing here. The *type* information is needed for *type inference systems*, as can be found in works like [Cart85, Mish85, Kata84]. However, type inference is beyond the scope of the system we are developing here.

b. The empty sequence has structure <>;

c. If s_1 represents the structure of object x_1, s_2 represents the structure of object x_2, ..., s_n represents the structure of object x_n, then

$<s_1, s_2, \cdots, s_n>$ represents the structure of the sequence $<x_1, x_2, \cdots, x_n>$.

d. *Homogeneous sequences*: If every element x_i of a sequence $<x_1, x_2, \cdots, x_n>$ has the same structure s, we can define a more compact representation for the structure of the sequence. Two representations are defined. *Representation 1* is simply $<s^n>$. *Representation 2* captures more information than the previous one. It *enumerates* the structure of the elements of the homogeneous sequence: $<s^{1:n}>$. Note the difference between $<s_1, s_2, \cdots, s_n>$ and $<s^{1:n}>$. In the former case, the structure of each element of the sequence may be different, whereas in the later case all elements have the same structure. In Representation 2, if we want to single out one element of the sequence (say, the k^{th} element) we use $s^{k:k}$. This is to remove ambiguity between this case and s^k, which represents a list of k elements with same structure under Representation 1. The usefulness of the distinction between the two representations will soon become clear.

Example: The input for a matrix multiplication program, consisting of the sequence of a matrix $A_{n\times m}$ and a matrix $B_{m\times l}$, has the following structure representation: $<<<a^m>^n> \ < <a^l>^m> >$. Since this case presents homogeneous sequences, we can use the alternative representation: $<<<a^{1:m}>^{1:n}> \ < <a^{1:l}>^{1:m}> >$.

The distinctive treatment given to *homogeneous sequences* will be of foremost importance in practice. It will allow the compiler to detect and efficiently manipulate such sequences, which are nothing more than regular

structures (vectors and arrays). As for the two alternative representations, we will sometimes want to capture more detail and sometimes less. For example, if we apply the primitive **tl** to a sequence of structure $<s^n>$ the result will have structure $<s^{n-1}>$. Similarly, **tlr** applied to $<s^n>$ also has a result with structure $<s^{n-1}>$. Only the alternative representation captures the distinct behaviors of **tl** and **tlr**. The primitive **tl** applied to $<s^{1:n}>$ results in $<s^{2:n}>$; whereas **tlr** applied to $<s^{1:n}>$ results in $<s^{1:n-1}>$.

If we pose some restrictions on the structure of input objects we can describe, for each FP primitive, the structure of the expected result object. Now we describe the basic *structural transformations* induced by some of the FP primitives. The complete description can be found in Appendix 2. In the description, the notation $f : s \rightarrow t$ means that an FP function f applied to an object of structure s returns an object of structure t. Note the *restrictions* imposed on the primitives **distl, distr, trans, pair** and **split**; they will be analyzed in the next section.

Selectors: **k**: $<s_1, s_2, \cdots, s_n>$ and $1 \le k \le n \rightarrow s_k$; Λ

For homogeneous sequences:

k: $<s^{1:n}>$ and $1 \le k \le n \rightarrow s^{k:k}$; Λ

last: $<s_1, s_2, \cdots, s_n>$ and $n \ge 1 \rightarrow s_n$; Λ

For homogeneous sequences:

last: $<s^{1:n}>$ and $n \ge 1 \rightarrow s^{n:n}$; Λ

tl: $<s_1, s_2, \cdots, s_n>$ and $n \ge 2 \rightarrow <s_2, s_3, \cdots, s_n>$; Λ

For homogeneous sequences:

tl: $\langle s^{1:n} \rangle$ and $n \geq 2 \rightarrow \langle s^{2:n} \rangle$; Λ

distl: $\langle s, \langle t^{1:n} \rangle \rangle \rightarrow \langle \langle s, t \rangle^{1:n} \rangle$; Λ

Restriction: Second element is a homogeneous sequence.

distr: $\langle \langle s^{1:n} \rangle, t \rangle \rightarrow \langle \langle s, t \rangle^{1:n} \rangle$; Λ

Restriction: First element is a homogeneous sequence.

apndl: $\langle s, \langle t_1, t_2, \cdots, t_n \rangle \rangle \rightarrow \langle s, t_1, t_2, \cdots, t_n \rangle$; Λ

For homogeneous sequences:

$\langle s, \langle t^{1:n} \rangle \rangle \rightarrow \langle s, t^{1:n} \rangle$; Λ

Special case:

$\langle t, \langle t^{1:n} \rangle \rangle \rightarrow \langle t^{1:n+1} \rangle$; Λ

trans: $\langle \langle s^{1:m} \rangle^{1:n} \rangle \rightarrow \langle \langle s^{1:n} \rangle^{1:m} \rangle \quad m, n \geq 1$; Λ

Restriction: Homogeneous sequences.

pair: $\langle s^n \rangle \rightarrow \langle \langle s^2 \rangle^{n/2} \rangle$; Λ

Restrictions: Homogeneous sequence and n even.

split: $\langle s^{1:n} \rangle \rightarrow \langle \langle s^{1:n/2} \rangle \langle s^{n/2+1:n} \rangle \rangle$; Λ

Restrictions: Homogeneous sequence and n even.

eq: $\langle s, t \rangle \rightarrow \begin{cases} \mathbf{F}, & \text{if } s \neq t \\ a, & \text{if } s = t \end{cases}$

null: $\langle \rangle \rightarrow$ **T; F**

length: $\langle s_1, s_2, \cdots, s_n \rangle \rightarrow n$; $\langle \rangle \rightarrow 0$; Λ

For homogeneous sequences:

length: $<s^{1:n}> \rightarrow n; \Lambda$

+, -, *, /, and, or: $<a^2> \rightarrow a; \Lambda$

iota: $a \rightarrow <a^n>$, where $n>0$, n integer; Λ

In general, n is indeterminate. However, **iota** can be solved if applied to a (compile time) constant:

iota @ %3: x ≡ <1 2 3>

iota @ length: <a b c> ≡ <1 2 3>

2.1.3 Structural Behavior of Functional Forms

Below is the description of the structural behavior of the FP functional forms.

Composition

$$\sigma\,(f@g:s) \equiv \sigma\,(f:\sigma\,(g:s))$$

Construction

$$\sigma\,([f_1, f_2, \cdots, f_n]:s) \equiv <\sigma(f_1:s), \sigma(f_2:s), \cdots, \sigma(f_n:s)>$$

Apply-to-All

$$\sigma\,(\&f:<s_1, s_2, \cdots, f_n>) \equiv <\sigma(f:s_1), \sigma(f:s_2), \cdots, \sigma(f:s_n)>$$

Very often in FP programs, apply-to-all is used over a homogeneous sequence, i.e., in the form $\&f:<s^{1:n}>$. Unfortunately, it is *not* true that if objects x, y have the same structure then $f:x$, $f:y$ will have the same structure. An easy counter-example is:

&iota: <1, 2, 3> ≡ <<1>, <1, 2>, <1, 2, 3>>.

However, there is a significant class of FP functions where it is true that if objects x, y have the same structure then $f:x$, $f:y$ will have the same structure. Formally, we have:

Definition: The *class of structurally well-behaved* FP functions is formed by those functions f such that

$$\text{if } \sigma(x) \equiv \sigma(y) \text{ then } \sigma(f:x) \equiv \sigma(f:y),$$

where x, y are objects. In particular, all primitive FP functions, except for **iota**, and with the restrictions imposed in the definitions of the previous pages, are structurally well-behaved. In the next section, we study how often these well-behaved functions occur in FP programs.

For the class of structurally well-behaved functions and for homogeneous sequences, we have the following behavior for apply-to-all:

$$\sigma\,(\&f:<s^{1:n}>) \equiv <\sigma^{1:n}(f:s)>$$

Constant

$\%\, x : y \equiv x$, for all objects x, y.

Conditional

As pointed out before, the conditional functional form can be considered one of the major obstacles to a complete structural evaluation of FP programs. Basically, whenever the structure of the output object depends on the *value* of the input object the result of a symbolic evaluation system is non-deterministic. Therefore, we restrict the conditional as follows.

Two types of conditionals are permitted. The first type act as a switch. For this type, $(p \rightarrow f ; g)$, f and g must produce structurally equivalent output objects for any input object. For example, in

(> @ [1, 2] -> 1; 2)

the outcome of **(> @ [1, 2])** depends on the value of the input object (which is supposed to be a list of at least two numbers). However, independent of the result of the predicate, the final result of the function has the same structure (in this case, an atom). Formally, we allow all conditionals $(p \rightarrow f ; g) : x$ where $\sigma(f : x) \equiv \sigma(g : x)$ for all x.

The second type of conditional can be interpreted as structural control. The predicate must be based purely on structure (e.g., **atom**, **null**, **= @ [length, %5]**). The value of the predicate can be determined by the structure of the input object. In this case, the structure of the result will be the structure of one of the two functions (f or g) applied to the input, depending on the

value (T or F) obtained from applying p to the structure of the object. By allowing this second type of conditional, all recursions that are terminated by a structural predicate can be unfolded completely. This allows a whole class of computations that can be represented by acyclic computational graphs to be structurally evaluated.

Right Insert, Tree Insert

We treat both inserts uniformly. We restrict them so that they act upon homogeneous sequences and are applied to the following FP primitive functions only: +, -, *, /, **and, or, xor**. Then, we have:

$$\sigma(!f : <a^{1:n}>) \equiv a$$

$$\sigma(|f : <a^{1:n}>) \equiv a$$

2.2 Analysis of the Restrictions

In this section we will analyze a reasonable set of FP programs to determine the percentage of real programs that do not fall in the class defined by the imposed restrictions. We recall that we imposed restrictions on the input object structure for the FP primitives **distl, distr, trans, pair** and **split**. Also, the conditional functional form and the inserts functional forms are restricted. It is also worth of note that the functional form **while** was not allowed.

A sample of 49 FP programs was collected for the analysis. They are distributed according to the following sources:

a. 8 from [Schl84]: The programs were developed to demonstrate an FP approach in the layout of VLSI circuits. This sample is somewhat biased since the conditional functional form is restricted in that work; therefore only predicates based on the structure are found on the 8 FP programs.

b. 16 FP programs developed by us for the manipulation of graphs. Graphs are represented by their adjacency matrix. The most elaborate program computes the transitive closure of a graph.

c. 16 FP programs for parallel associative searching from [Will81].

d. 9 other programs from varied sources.

A total of 176 FP function definitions was present in the sample (49 definitions for the programs plus 127 definitions for auxiliary functions used by the programs). Table 2.1 shows the static counts for the the 791 occurrences of FP primitives in the sample.

It is important to notice the high incidence of structural primitives in the sample (83.57% of the total). Another indication of the importance of structural primitives in FP can be seen in Table 2.2 which shows the ten most used primitives (which accounts for 74.21% of all occurrences). Notice that eight of them are structural; furthermore, only the selectors **1**, **2** and **3** account for almost 43% of the total.

Table 2.3 shows the static frequency of occurrence for the functional forms. Notice that the **while** functional form did not appear at all in the sample

Structural Primitives			Computational Primitives		
primitive	freq.	%	primitive	freq.	%
1	167	21.11	iota	7	0.88
2	152	19.22	pick	8	1.01
3	21	2.65	+	13	1.64
other selectors	1	0.13	-	6	0.76
id	49	6.19	*	11	1.39
first	1	0.13	/	1	0.13
last	19	2.40	mod	1	0.13
tl	22	2.78	not	8	1.01
tlr	13	1.64	nand	0	0.00
distl	30	3.79	nor	0	0.00
distr	15	1.90	and	26	3.27
trans	43	5.44	or	12	1.52
pair	16	2.02	xor	2	0.25
split	16	2.02	eq	25	3.16
concat	52	6.57	neq	0	0.00
apndl	12	1.52	lt	2	0.25
apndr	7	0.88	le	2	0.25
length	17	2.15	gt	2	0.25
atom	3	0.38	ge	4	0.51
null	3	0.38			
reverse	2	0.25			
rotl	0	0.00			
rotr	0	0.00			
Totals	661	83.57	Totals	130	16.43
			TOTAL	791	100.00

Table 2.1 - Static Frequency of FP Primitives

examined.

The static counts presented in the these tables are useful to show how often the opportunities for the use of structural evaluation occur in FP programs. However, static analysis is not sufficient to reveal whether an occurrence of an FP primitive is in accordance with the restrictions imposed in the previous section. In order to analyze those restrictions, we studied the structure of the input object and of the intermediate objects, the results of

primitive	freq.	%
1	167	21.11
2	152	19.22
concat	52	6.57
id	49	6.19
trans	43	5.44
distl	30	3.79
and	26	3.27
eq	25	3.16
tl	22	2.78
3	21	2.65
Total	587	74.21

Table 2.2 - Ten Most Used Primitives

functional form	freq.	%
composition	595	53.89
construction	248	22.46
apply-to-all	143	12.95
constant	58	5.25
conditional	40	3.62
inserts	20	1.81
while	0	0.00
Total	1104	100.00

Table 2.3 - Static Frequency of Functional Forms

which are summarized in the list below:

a. **distl**: two of the 30 occurrences did not have the second element as an homogeneous sequence;

b. **distr**: one of the 15 cases was not in accordance with the restriction (first element must be an homogeneous sequence);

c. **trans**: three out of the 43 cases violated the homogeneous sequences restriction;

d. **pair**: four of the 16 occurrences did not have an homogeneous sequence as input object;

e. **split**: two of the 16 cases violated the homogeneous sequence assumption;

f. conditional: 25 of the 40 cases had predicates solely based on structure according to the restriction; 8 of the remaining cases had predicate based on value, but had a predictable output structure, thereby in accordance with the restrictions. The 7 remaining cases must be considered unsolvable by the restrictions outlined in the previous section.

g. inserts: all cases had homogeneous sequences as input objects; however 4 of the 20 cases did not have the required FP primitives as the functional parameter.

h. apply-to-all: *all* cases had homogeneous sequences as input objects. Furthermore, *all* cases belonged to the class of structurally well-behaved functions as defined previously.

A final and important analysis was made of the distribution of the violating occurrences over the 49 programs. This is important because we consider a program as not completely amenable to solution by the system defined above when a single restriction is violated. At first sight, the results were not very encouraging: only 23 of the 49 programs did not present any violation of the restrictions. A closer look showed that 14 of the 16 associative search programs were deemed unsolvable because of two defined FP functions, min and index, which presented conditionals based on value. However, as was discussed in Section 2.1.1, it is possible to solve the program partially, until a non-solvable primitive or functional form is found. For those 14 pro-

grams, it was found that min and index are used in the programs only after a fair amount of data replication and movement that can be structurally solved. Therefore, most of the data manipulating functions can be solved also for these programs.

Although 23, or 37, out of 49 seems to be a bad result, if we look at the total of 176 FP function definitions, which includes the definitions of the programs and of the auxiliary functions, we find that 145 of the 176 are solvable. This is a slightly better result and also an encouraging one, since modularity is expected in the development of FP programs and can be exploited by a symbolic structural evaluator.

2.3 Using Structural Evaluation: Examples

Below we present two examples of structural evaluation: a matrix multiplication program (MM) and a Fast-Fourier Transform program (fftstages).

2.3.1 Matrix Multiplication

The following program computes the product of two conformable matrices of any size. It was taken from [Back78].

{MM &&(!+) @ &&&* @ &&trans @ &distl @ distr @ [1, trans@2] }

The structure of the input for this program is:

$<<<a^m>^n>\ <\ <a^l>^m>\ >$

The structural evaluation begins with the construction:

1: $<<<a^m>^n> \; < <a^l>^m> > \rightarrow < <a^m>^n>$

2: $<<<a^m>^n> \; < <a^l>^m> > \rightarrow < <a^l>^m>$

trans: $< <a^l>^m> > \rightarrow < <a^m>^l> >$

Then:

[1, trans@2]: $<<<a^m>^n> \; < <a^l>^m> > \rightarrow <<<a^m>^n> \; < <a^m>^l> >$

distr: $<<<a^m>^n> \; < <a^m>^l> > \rightarrow <<<a^m> < <a^m>^l>>^n>$

&distl: $<<<a^m> < <a^m>^l>>^n> \rightarrow <<<<a^m> <a^m>>^l>^n>$

$\equiv <<<<a^m>^2>^l>^n>$

&&trans: $<<<<a^m>^2>^l>^n> \rightarrow <<<<a^2>^m>^l>^n>$

&&&+: $<<<<a^2>^m>^l>^n> \rightarrow <<<a^m>^l>^n>$

&&(!+): $<<<a^m>^l>^n> \rightarrow < <a^l>^n>$ □

2.3.2 Fast-Fourier Transform

Now, we analyze a program taken from [Schl84]. It computes the Fast-Fourier Transform of a set of complex numbers, except for the final bit-reversal stage. The size of the set can be any power of 2.

```
{fftstages    (= @ [length, %2] -> W;
                                   concat @ &fftstages @
                                   split @ concat @ Bfly @ concat @
                                   &W @ Bfly
              )
}
```

```
{Bfly  concat @                              # butterfly permutation
       [trans @ [1,3], trans @ [2,4]] @
       concat @ &trans @ split @ pair}

{cadd &+ @ trans}   # complex addition
{csub &- @ trans}   # complex subtraction
{cmul [ - @ [ *@[1@1,1@2], *@[2@1,2@2] ],  # complex multiply
        + @ [ *@[1@1,2@2], *@[2@1,1@2] ]
       ]
}
{u0  [%1, %1]}
{W   [cadd, csub] @ [1, cmul @ [2, u0]] }
```

The structure of the input for this program is:

$<<a^2>^n>$, where $n=2^k$, $k \geq 2$.

First, we show that **Bfly**: $<s^n> \rightarrow < <s^2>^{n/2}>$.

pair: $<s^n> \rightarrow < <s^2>^{n/2}>$.

split: $< <s^2>^{n/2}> \rightarrow <<<s^2>^{n/4}>^2>$.

&trans: $<<<s^2>^{n/4}>^2> \rightarrow <<<s^{n/4}>^2>^2>$.

concat: $<<<s^{n/4}>^2>^2> \rightarrow < <s^{n/4}>^4>$.

[1, 3]: $< <s^{n/4}>^4> \rightarrow < <s^{n/4}>^2>$.

[2, 4]: $< <s^{n/4}>^4> \rightarrow < <s^{n/4}>^2>$.

trans@[1, 3]: $< <s^{n/4}>^4> \rightarrow < <s^2>^{n/4}>$.

trans@[2, 4]: $< <s^{n/4}>^4> \rightarrow < <s^2>^{n/4}>$.

[trans@[1, 3], trans@]2, 4]]: $< <s^{n/4}>^4> \rightarrow <<<s^2>^{n/4}>^2>$.

concat: $<<<s^2>^{n/4}>^2> \rightarrow < <s^2>^{n/2}>$.

Now, we show that **W:** $< <a^2>^2> \rightarrow < <a^2>^2>$.

u0: $t \rightarrow <a^2>$.

[2, u0]: $< <a^2>^2> \rightarrow < <a^2>^2>$.

cmul: $< <a^2>^2> \rightarrow <a^2>$.

[1, cmul@[2, u0]]: $< <a^2>^2> \rightarrow < <a^2>^2>$.

cadd: $< <a^2>^2> \rightarrow <a^2>$.

csub: $< <a^2>^2> \rightarrow <a^2>$.

[cadd, csub]: $< <a^2>^2> \rightarrow < <a^2>^2>$.

Finally, we show by induction on k that

fftstages: $<s^{2^k}> \rightarrow <s^{2^k}>$, where $s \equiv <a^2>$.

Basis: (k=1) **fftstages:** $<s^2> \rightarrow <s^2>$.

> **[length, %2]:** $<s^2> \equiv <2,2>$.
>
> $=: <2,2> \equiv$ **T.**
>
> Therefore, the true branch of the conditional is applied and we have:
>
> **W:** $<s^2> \rightarrow <s^2>$, as showed before.

Hypothesis: **fftstages:** $<s^{2^k}> \rightarrow <s^{2^k}>$.

Step: **fftstages:** $<s^{2^{k+1}}>$

> **Bfly:** $<s^{2^{k+1}}> \rightarrow < <s^2>^{2^k}>$, as shown above.
>
> **&W:** $< <s^2>^{2^k}> \rightarrow < <s^2>^{2^k}>$, as shown above.
>
> **concat:** $< <s^2>^{2^k}> \rightarrow <s^{2^{k+1}}>$.

Bfly: $<s^{2^{k+1}}> \rightarrow < <s^2>^{2^k} >$, as shown above.

concat: $< <s^2>^{2^k} > \rightarrow <s^{2^{k+1}}>$.

split: $<s^{2^{k+1}}> \rightarrow < <s^{2^k}>^2 >$.

&fftstages: $< <s^{2^k}>^2 > \rightarrow < <s^{2^k}>^2 >$; by the induction hypothesis.

concat: $< <s^{2^k}>^2 > \rightarrow <s^{2^{k+1}}>$ □

2.4 Representation of Regular Structures in a Linear Memory

In this section we expand the algebra of structural transformations by imposing a memory model for the storage of objects. Assume a linear memory as the model for storage of objects. The input object to an FP program will be stored in the memory beginning at location 0 and occupying consecutive locations. We also assume that each memory location can hold any FP atom (numbers, characters, etc.). With this storage model, we can represent regular structures by using *algebraic equations* that describe the positions of each atom of the structure in the memory.

For example, the positions of the elements of a vector A[1:n] can be represented by the following equation:

$$loc(a_i) = i - 1, \quad 1 \le i \le n$$

Similarly, a matrix B[1:n, 1:m] would have the following equation:

$$loc(b_{ij}) = m(i-1)+(j-1), \quad 1 \le i \le n, \quad 1 \le j \le m$$

Clearly, similar equations can be derived to represent higher-order arrays.

Using these algebraic equations to represent regular structures, we can derive the algebraic equation of the expected result object for each FP structural primitive. Now we describe the basic *algebraic transformations* induced by the FP structural primitives.

Selectors

k: A[1:n], $1 \le k \le n$

input: $loc(a_i) = i-1, \quad 1 \le i \le n$

output: $loc(a_k) = 0$

k: A[1:n, 1:m], $1 \le k \le n$

input: $loc(a_{ij}) = m(i-1)+(j-1), \quad 1 \le i \le n, \quad 1 \le j \le m$

output: $loc(a_{kj}) = j-1, \quad 1 \le j \le m$

Similarly, for higher-order arrays.

Last

last: A[1:n]

input: $loc(a_i) = i-1, \quad 1 \le i \le n$

output: $loc(a_n) = 0$

last: A[1:n, 1:m]

input: $loc(a_{ij}) = m(i-1)+(j-1), \quad 1 \le i \le n, \quad 1 \le j \le m$

output: $loc(a_{nj}) = j-1, \quad 1 \le j \le m$

Similarly, for higher-order arrays.

Tail

tl: A[1:n]

input: $loc(a_i) = i-1, \quad 1 \le i \le n$

output: $loc(a_i) = i-2, \quad 2 \le i \le n$

tl: A[1:n, 1:m]

input: $loc(a_{ij}) = m(i-1)+(j-1), \quad 1 \le i \le n, \quad 1 \le j \le m$

output: $loc(a_{ij}) = m(i-2)+(j-1), \quad 2 \le i \le n, \quad 1 \le j \le m$

Similarly, for higher-order arrays.

Tail-Right:

tlr: A[1:n]

input: $loc(a_i) = i-1, \quad 1 \le i \le n$

output: $loc(a_i) = i-1, \quad 1 \le i \le n-1$

tlr: A[1:n, 1:m]

input: $loc(a_{ij}) = m(i-1)+(j-1), \quad 1 \le i \le n, \quad 1 \le j \le m$

output: $loc(a_{ij}) = m(i-1)+(j-1), \quad 1 \le i \le n-1, \quad 1 \le j \le m$

Similarly, for higher-order arrays.

Distribute-Left:

distl: <*s*, A[1:n]>

Distribute-left replicates n times the first argument s. Below, *sizeof*(s) is the number of memory cells used by object s.

input: $loc(a_i) = sizeof(s)+(i-1), \quad 1 \le i \le n$

output: $loc(a_i) = (i-1)+i \times sizeof(s), \quad 1 \le i \le n$

$loc(s)^{*k} = (k-1)+(k-1) \times sizeof(s), \quad 1 \le *k \le n$

Note the notation $*k$ to indicate that we have replication of object s.

If s itself is a vector:

distl: <S[1:m], A[1:n]>

input: $loc(s_i) = (i-1), \quad 1 \le i \le m$

$loc(a_i) = m+(i-1), \quad 1 \le i \le n$

output: $loc(s_i)^{*k} = m(k-1)+(i-1), \quad 1 \le i \le m, \quad 1 \le *k \le n$

$loc(a_i) = (i-1)+im, \quad 1 \le i \le n$

The extension to higher-order arrays is straightforward. Distribute-right (**distr**) behaves similarly; only the second element is the one to be replicated.

Transpose:

trans: A[1:n, 1:m]

input: $loc(a_{ij}) = m(i-1)+(j-1), \quad 1 \le i \le n, \quad 1 \le j \le m$

output: $loc(a_{ij}) = (i-1)+n(j-1), \quad 1 \le i \le n, \quad 1 \le j \le m$

trans: $<A_1[1:n], A_2[1:n], \cdots, A_m[1:n]>$

input: $loc(a_{1i}) = (i-1), \quad 1 \le i \le n$

$loc(a_{2i}) = n+(i-1), \quad 1 \le i \le n$

$\cdots$

$loc(a_{mi}) = (m-1)n+(i-1), \quad 1 \le i \le n$

output: $loc(a_{1i}) = m(i-1), \quad 1 \le i \le n$

$loc(a_{2i}) = 1+m(i-1), \quad 1 \le i \le n$

$\ldots$

$loc(a_{mi}) = (m-1)+m(i-1), \quad 1 \le i \le n$

The other structural FP primitives – **apndl**, **apndr**, **concat**, **pair** and **split** – do not move the atoms of the object; only the structure changes according to the transformations of Section 2.1.2. Therefore, there exists no change in the algebraic equations that describe the positions of regular structures when any of these FP primitives are applied.

2.5 Conclusion

The technique described above, besides giving an algebraic formulation for structural transformations in FP, allows a precise definition of the class of programs amenable to manipulation in the structural domain. Furthermore, it gives the foundations on which a compiler can be built. Indeed, the compiler is the practical side of the algebra just described. The compiler, presented in the next chapter, implements each of the basic relations described above; it follows, step by step, the same derivations shown in the two examples in order to generate an efficient run-time environment for FP programs.

CHAPTER 3
COMPILATION AND MEMORY MANAGEMENT

3.1 Manipulation of Algebraic Equations

In Chapter 2, an algebra for symbolic structural evaluation of FP programs was defined. In this chapter, we show how the algebra can be implemented as a compiler for FP programs. The compiler gathers information on the structure of the algorithm and of the input object, and solves the structural primitives *at compile time*. Figure 3.1 shows an overall scheme of the approach.

This approach will be clearly beneficial if the objects involved in the computation are of regular nature, such as vectors and matrices. However, general lists are also expected to take advantage of the method.

Below, we give an idea of how the compiler works by means of an example. A more detailed discussion on some implementation issues is given in the next section. The example used is the matrix multiplication program (MM) presented in the previous chapter.

We assume the same linear memory model for storage of objects presented in Section 2.4. The input object to the FP program is stored in the memory beginning at location 0 and occupying consecutive locations. Note

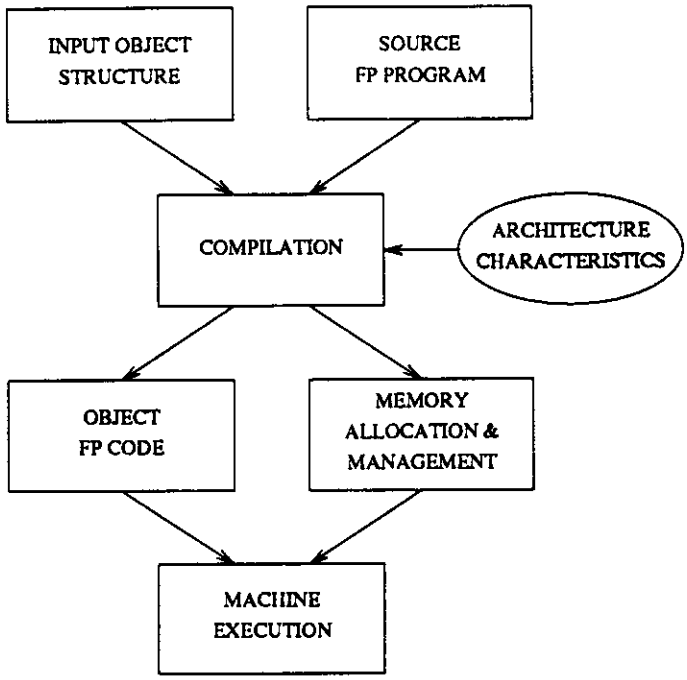


Figure 3.1 - Compilation: Proposed Approach for Optimization of FP Programs

that the assumption of linear memory does not imply that the processor is sequential or of the von Neumann type. For example, Magó's machine [Mago80] is a full binary tree of processors where the memory can be considered to be linear (the leaves of the tree constitute the memory of the machine).

Suppose the input object is a list with matrices A(n x m) and B(m x l). Note that the first four steps of the program have only structural primitives (**&&trans @ &distl @ distr @ [1, trans@2]**). What this part does is to manipulate and expand both matrices so they become three-dimensional objects interleaved element by element in a form appropriate for all the multiplications to be done in just one step. For clarity, we rewrite MM by dividing

it in a structural part and a computational part:

{MM compute @ expand }

{compute &&(!+) @ &&&* }

{expand &&trans @ &distl @ distr @ [1, trans@2] }

If we store A and B in the linear memory as described above, we can identify any element of either matrix by the following algebraic equations (Figure 3.2 illustrates the matrices' elements positions for n=2, m=3, l=4):

$$loc(a_{ij}) = m(i-1)+(j-1),\ 1 \le i \le n,\ 1 \le j \le m \tag{3.1a}$$

$$loc(b_{ij}) = nm+l(i-1)+(j-1),\ 1 \le i \le m,\ 1 \le j \le l \tag{3.1b}$$

After the first step of *expand* ([**1, trans@2**]), matrix A does not change and matrix B is transposed; this new situation can be described by the equations:

$$loc(a_{ij}) = m(i-1)+(j-1),\ 1 \le i \le n,\ 1 \le j \le m \tag{3.2a}$$

$$loc(b_{ij}) = nm+(i-1)+m(j-1),\ 1 \le i \le m,\ 1 \le j \le l \tag{3.2b}$$

The primitive **distr** broadcasts one copy of matrix B to the right of each row of matrix A. The new positions are described by:

$$loc(a_{ij}) = (m+ml)(i-1)+(j-1),\ 1 \le i \le n,\ 1 \le j \le m \tag{3.3a}$$

$$loc(b_{ij}^{k}) = m+(i-1)+m(j-1)+(m+ml)(k-1),\ 1 \le i \le m,\ 1 \le j \le l,\ 1 \le^{*} k \le n \tag{3.3b}$$

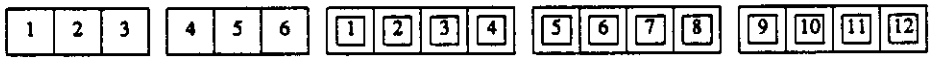
$$\begin{bmatrix}1 & 2 & 3\\ 4 & 5 & 6\end{bmatrix} \times \begin{bmatrix}1 & 2 & 3 & 4\\ 5 & 6 & 7 & 8\\ 9 & 10 & 11 & 12\end{bmatrix}$$

A x B

Elements of A are inside a □

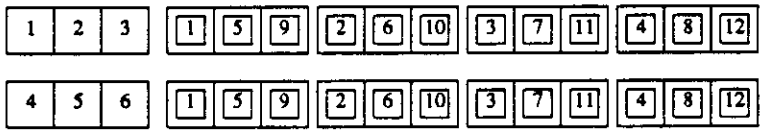
Elements of B are inside a □

Initial positions:



After **[1, trans@2]**:

After **distr**:



After **&distl**:

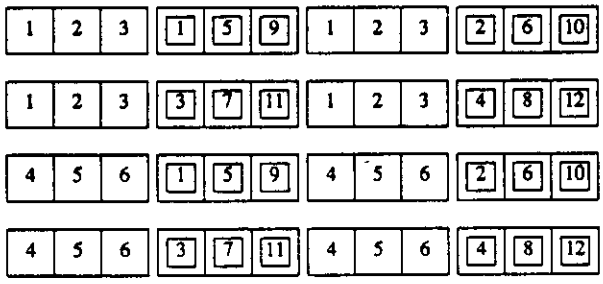
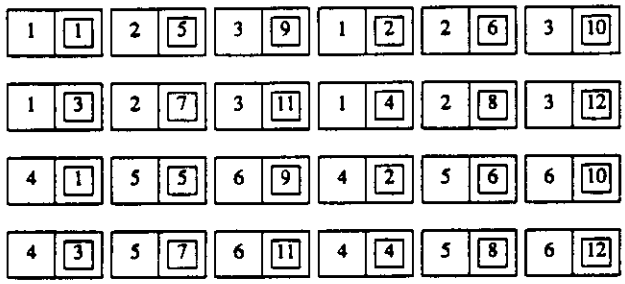


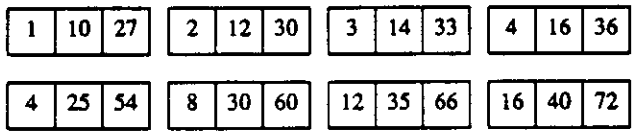
Figure 3.2 - MM Example for A(2 x 3) and B(3 x 4)

(n=2, m=3, l=4)

After **&&trans:**



After **&&&*** (new object C):



After **&&(!+)** (final object D):

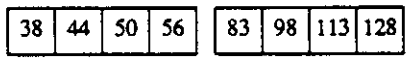


Figure 3.2 (cont'd) - MM Example for A(2 x 3) and B(3 x 4)

Note that these new equations reflect the following facts: a) each row of A is separated by strides of $(m+ml)$, i.e., the size of each line of A plus the size of each copy of B; b) a new index k (from 1 to n, the number of lines of A) represents the multiple copies of B; it is marked with a * to show that it represents repetitions of the original object as was done in Section 2.4; c) the first copy of matrix B now begins at location m, just after the first line of A.

The next step, **&distl**, broadcasts one copy of each line of A to each line of each copy of B (transposed). The resulting structure has n lists of l lists of 2 lists of m elements each and can be described by the following equations:

$$loc(a_{ij}^{k}) = 2ml(i-1)+(j-1)+2m(k-1),\ 1\le i\le n,\ 1\le j\le m,\ 1\le^{*} k\le l \qquad (3.4a)$$

$$loc(b_{ij}^{k}) = m+(i-1)+2m(j-1)+2ml(k-1),\ 1\le i\le m,\ 1\le j\le l,\ 1\le^{*} k\le n \qquad (3.4b)$$

Finally, the last step of *expand* (**&&trans**) yields the following equations:

$$loc(a_{ij}^{k}) = 2ml(i-1)+2(j-1)+2m(k-1),\ 1\le i\le n,\ 1\le j\le m,\ 1\le^{*} k\le l \qquad (3.5a)$$

$$loc(b_{ij}^{k}) = 1+2(i-1)+2m(j-1)+2ml(k-1),\ 1\le i\le m,\ 1\le j\le l,\ 1\le^{*} k\le n \qquad (3.5b)$$

Now the system must treat the computational part of the program, i.e., the function **compute**. The n^3 multiplications (**&&&***) are applied on the atoms described by the set of equations 3.5 and generate a new object that we call C and that has its positions described by the following equations:

$$loc(c_{ijk}) = ml(i-1)+2(j-1)+(k-1),\ 1\le i\le n,\ 1\le j\le l,\ 1\le k\le m \qquad (3.6)$$

The final step of the program takes the newly created object C and generates a new one, D, which is the final answer and occupies the positions described by:

$$loc(d_{ij}) = l(i-1)+(j-1),\ 1\le i\le n,\ 1\le j\le l \qquad (3.7)$$

After these transformations, a pseudo-code for the compiled MM program would have the following text:

1. Transfer elements of A from positions

   $loc(a_{ij}) = m(i-1)+(j-1),\ 1\le i\le n,\ 1\le j\le m$

   to positions

$$loc(a_{ij}^{k}) = 2ml(i-1)+2(j-1)+2m(k-1),\ 1\le i\le n,\ 1\le j\le m,\ 1\le k\le l$$

2. Transfer elements of B from positions

$$loc(b_{ij}) = nm+l(i-1)+(j-1),\ 1\le i\le m,\ 1\le j\le l$$

to positions

$$loc(b_{ij}^{k}) = 1+2(i-1)+2m(j-1)+2ml(k-1),\ 1\le i\le m,\ 1\le j\le l,\ 1\le k\le n$$

3. Multiply A*B generating result C in positions

$$loc(c_{ijk}) = ml(i-1)+m(j-1)+(k-1),\ 1\le i\le m,\ 1\le j\le l,\ 1\le k\le n$$

4. Add (multiple-operand) generating result D in positions

$$loc(d_{ij}) = l(i-1)+(j-1),\ 1\le i\le n,\ 1\le j\le l$$

With a system that, given the initial FP program and the structure of its input object, is able to make the transformations described by the set of equations from (3.1) to (3.7), without manipulating the real data, the following immediate benefits would result:

a. unnecessary data replication is eliminated in the step [1, **trans@2**], as discussed in Chapter 1;

b. data movement is minimized, since all the structural part of the program, which was composed by four steps, is collapsed to one step;

c. the equations have information about the amount of replication of each object; the compiler can restrain replication if the target machine is not sufficiently concurrent for the input object. Note also that the algorithm

implies no sequencing whatsoever; this is now left to the compiler, again in case of not sufficient parallelism;

d. the elimination of intermediate steps in the computation can reduce significantly the cost of garbage collection in systems that use this technique to reclaim storage.

Note that the method does not eliminate data movement completely; it simply brings together several steps of data movement and replication into a single step; in the above example, all steps of *expand* are abstracted by the transformation from equations (3.1) into equations (3.5).

It is clear that this approach needs structural information about the input object such as arrays dimensions. Although FP programs can be built that work for general structures (in reality, the above MM works for any conformable matrices), the fact that the user has to supply information on structure and size is not necessarily bad or less general since the programmer knows anyway what will be the kind of input object the function is expected to act upon. Furthermore, this information will be needed by the machine sooner or later; what we are proposing here is to have the information sooner and take advantage of it to improve the overall performance of the system.

The next section discusses some implementation issues of the ideas developed in this section for the case of an uniprocessor system.

3.2 Uniprocessor Implementation

3.2.1 General Structure of the System

A compiler was built to implement the algebraic transformations outlined in the previous section. It is written in the C programming language [Kern78] and uses the programs Lex [Lesk75] and Yacc [John75] of the Unix† operating system for the lexical and syntactical analysis of FP programs. It accepts as input FP programs together with the structural description of the input object. Figure 3.3 presents the overall structure of the compiler. The syntax is the same of Berkeley-FP as described in [Bade83], except for the introduction of the description of objects' structures. Basically, the user has a shortened way to describe the structure of vectors and matrices. For example, the way to describe a list consisting of two 10×10 matrices as input to an FP function **MM** is to write **MM**: <A(10,10), B(10,10)>.

Figure 3.3 reveals that the compilation phases seem rather conventional. This is indeed the case up to the syntactical analysis. As the parser scans the input FP program, a syntax (program) tree representing the program is built. Later, in the intermediate code generation phase, this tree is traversed while code is being generated. It is the intermediate code generation phase that truly differentiates this compiler from straight compilation; at this phase structural transformations are done on the FP program using *object descriptors*, as described in the following sections.

† Unix is a trademark of AT&T

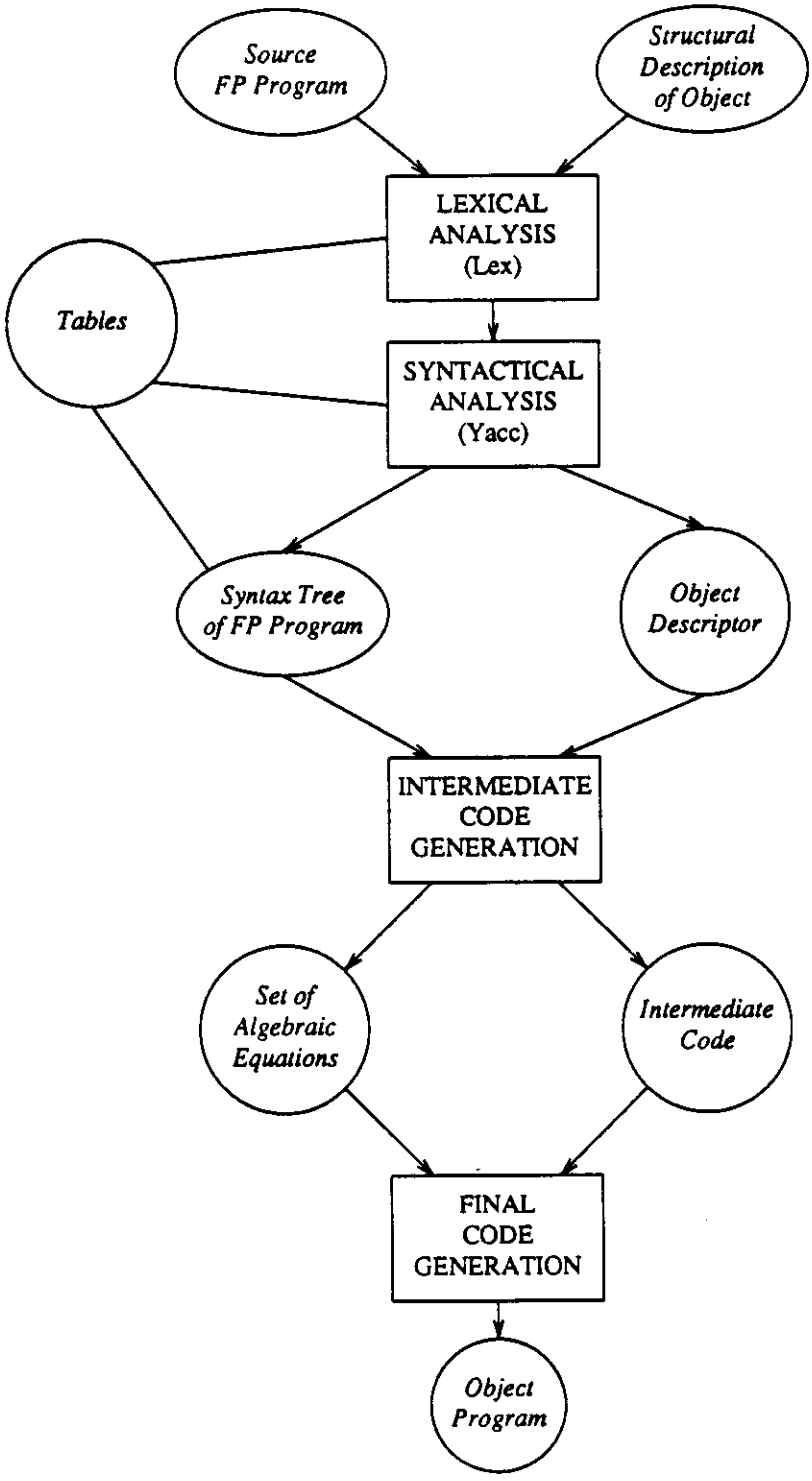


Figure 3.3 - Compiler Structure

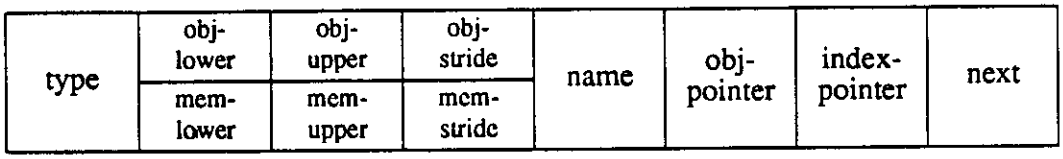
3.2.2 Object Descriptors

The most important data structure of the compiler, which enables the manipulation of the algebraic equations, is the *object descriptor*. At the moment of the definition of the descriptors we must make fundamental assumptions on the architecture of the underlying machine. This implementation, as mentioned in the previous section, assumes a linear memory of contiguous cells each capable of storing one FP atom.

The motivations behind the existence of these data structures is based on the observation that it is cheaper to manipulate *descriptors* of regular data structures than the data themselves. For example, an array transposition might be implemented by exchanging indexes in the array descriptor.

The descriptor (see Figure 3.4) keeps information such as: a) structure of the object; b) lower and upper limits of data such as arrays and vectors; c) lower and upper memory location of objects; d) strides between different portions of the same object; e) indication of repetition of parts of objects; f) index names. The initial descriptor is built by the compiler when it reads the input structure of the object. At the intermediate code generation phase, the compiler executes the structural primitives on the descriptor, generating a new descriptor that reflects the effect of the primitives. This phase is very much like a *symbolic execution*; the difference is that the compiler needs only information on the structure of the input object, not the object itself. This process continues until all possible transformations are done, that is, it is possible to have a partial transformation of the original program; when the compiler finds an FP primitive or functional form that cannot be structurally solved, it

continues compilation but without making the structural transformations.



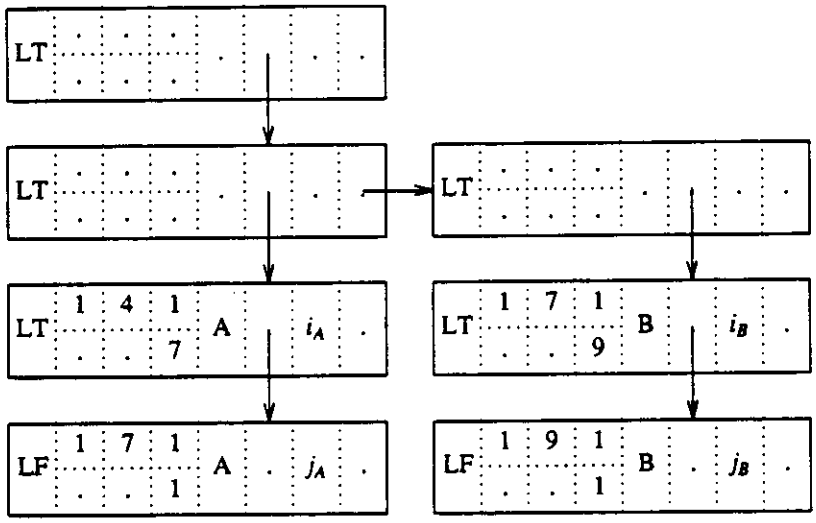

```
struct objnode {
        int type;                       /* see below list of types                    */
        int objlower;                   /* lower limit for arrays                     */
        int objupper;                   /* upper limit for arrays                     */
        int objstride;                  /* stride between array elements              */
        int memlower;                   /* lower memory position allocated to array   */
        int memupper;                   /* upper memory position allocated to array   */
        int memstride;                  /* memory stride for array elements           */
        struct valuenode *name;         /* pointer to name of object                  */
        struct objnode  *objpointer;    /* pointer to object description at lower level*/
        struct indexnode *indexpointer; /* pointer to list of indices of array object */
        struct objnode  *next;          /* pointer to next object at same level       */
};

type ∈ { LIST, LEAF, PAIRED }
```

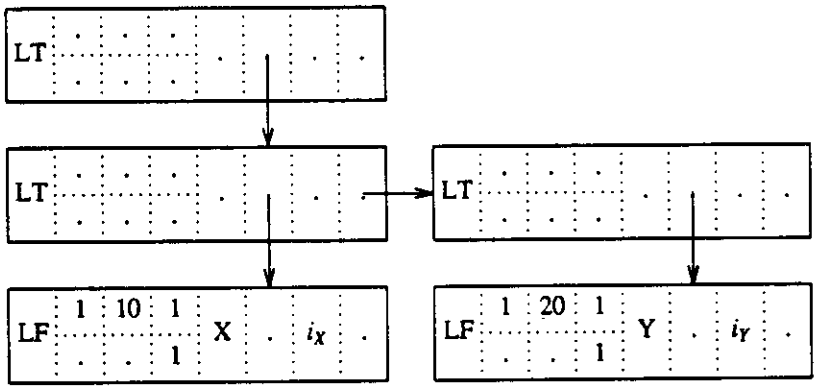
Figure 3.4 - Object Descriptor Nodes

The descriptor of an FP object, built when the description of the input object is read by the syntactical analyzer, and manipulated afterwards by the intermediate code generator, has the structure of a tree, linked by the *objpointer* and *next* fields. Figure 3.5 illustrates the complete object descriptor for two examples of FP input objects.

By traversing the object descriptor, it is possible to generate the algebraic equations that describe the positions of the elements of the object. The object descriptor captures the structure of the object as well as the location of its elements (fields *objlower, objupper, objstride, memlower, memupper, memstride*).



(a) < A(4,7), B(7,9) >



(b) < X(10), Y(20) >

Figure 3.5 - Examples of Object Descriptors

3.2.3 Intermediate Code Generation

This phase solves the FP structural primitives with the restrictions outlined in Chapter 2 and translates the FP program when a non-structural primitive is found. All functions inside a construction and all applications of an apply-to-all are examined. The output of the compiler is the reduced FP program along with object descriptors that can be reduced to algebraic equations describing the positions of the atoms in the new objects.

The algorithm assumes that the FP program is in accordance with all the restrictions described in Chapter 2. When a restriction is encountered, translation continues but no more structural transformations are done. The algorithm adopts a string reduction semantics; for each FP primitive a copy of the descriptor is passed to the algorithm that implements that primitive. In other words, there is no sharing of descriptors.

The algorithms that compile each FP primitive begin by checking the consistency of the input object structure. This checking phase implements the semantics of the structural bottom, Λ, as was defined in Chapter 2. Therefore, errors such as binary numeric functions – +, –, *, etc. – applied to structures not equal to a list of two atoms are caught here.

Each algorithms is logically divided into a few number of cases that reflect the definitions of the structural transformations of Chapter 2. For example, the primitive **last** has two cases: the first for generic sequences and the second for homogeneous sequences. The final portion of each algorithm catches the violation of restrictions: if an object is structurally correct but did

not fall in the previous cases, the primitive cannot be solved by the compiler and is passed unsolved to the execution phase.

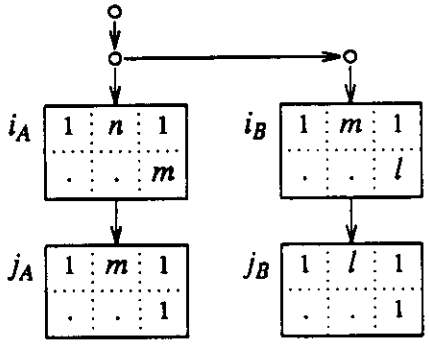
Figure 3.6 shows the evolution of the descriptor during the compilation of the matrix multiplication program. Since various nodes of type LIST often have undefined or null fields we use a more compact picture to represent object descriptors in Figure 3.6, as opposed to the full representation of Figure 3.5. Also note that the algebraic equations are not part of the descriptors, but are extracted from the descriptors by traversing them.

The final result of the intermediate code generation phase for an FP program is a sequence of pseudo-commands with associated descriptors. In the matrix multiplication example, the descriptors depicted in Figure 3.6 (a), (e), (f) and (g) are kept as part of the intermediate code. The pseudo-code for the compiled MM program has the text that was presented in Section 3.1.

That pseudo-code can be easily translated to a real uniprocessor assembler code by using the equivalent of **for** loops; all the information necessary to implement the loops is contained in the equations.

3.3 Other Realization Aspects

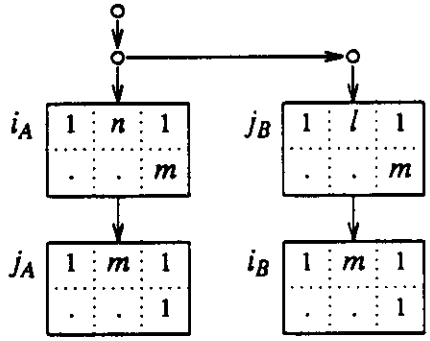
Compilation offers a wide variety of opportunities for code optimization of conventional languages [Aho86]. Many of the optimization techniques used in compiler construction for procedural languages can also be used in the



$$loc(a_{ij})=m(i-1)+(j-1), \quad 1\le i\le n,\ 1\le j\le m$$

$$loc(b_{ij})=nm+l(i-1)+(j-1), \quad 1\le i\le m,\ 1\le j\le l$$

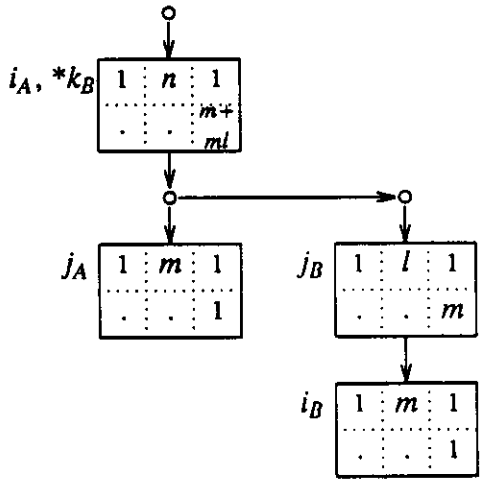
(a) Initial Object



$$loc(a_{ij})=m(i-1)+(j-1), \quad 1\le i\le n,\ 1\le j\le m$$

$$loc(b_{ij})=nm+(i-1)+m(j-1), \quad 1\le i\le m,\ 1\le j\le l$$

(b) After **[1, trans @ 2]**

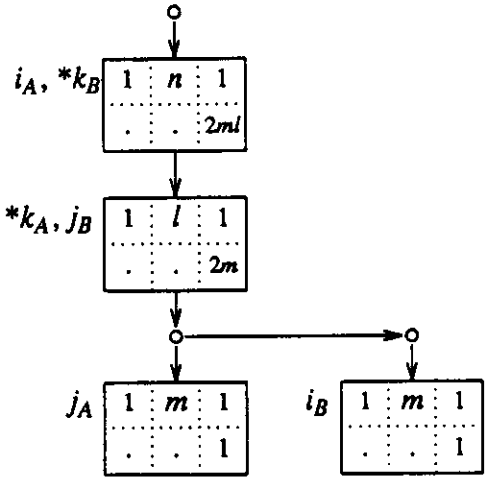


$$loc(a_{ij})=(m+ml)(i-1)+(j-1), \quad 1\le i\le n,\ 1\le j\le m$$

$$loc(b^k_{ij})=m+(i-1)+m(j-1)+(m+ml)(k-1), \quad 1\le i\le m,\ 1\le j\le l,\ 1\le *k\le n$$

(c) After **distr**

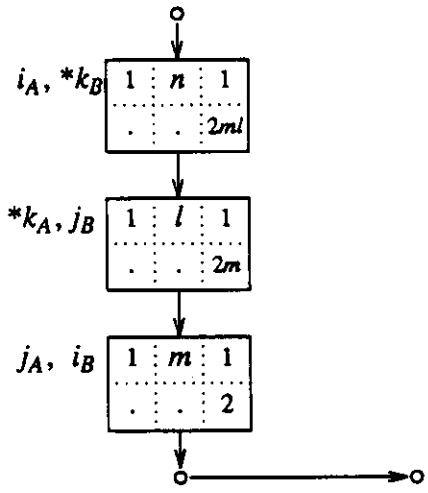
Figure 3.6 - Intermediate Code Generation for MM



$$loc(a_{ij}^k)=2ml(i-1)+(j-1)+2m(k-1), \quad 1\le i\le n,\ 1\le j\le m,\ 1\le *k\le l$$

$$loc(b_{ij}^k)=m+(i-1)+2m(j-1)+2ml(k-1), \quad 1\le i\le m,\ 1\le j\le l,\ 1\le *k\le n$$

(d) After **&distl**

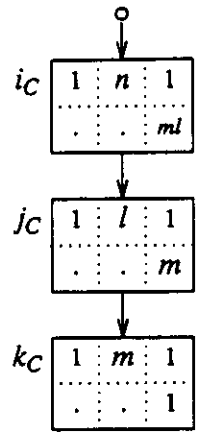


$$loc(a_{ij}^k)=2ml(i-1)+2(j-1)+2m(k-1), \quad 1\le i\le n,\ 1\le j\le m,\ 1\le *k\le l$$

$$loc(b_{ij}^k)=1+2(i-1)+2m(j-1)+2ml(k-1), \quad 1\le i\le m,\ 1\le j\le l,\ 1\le *k\le n$$

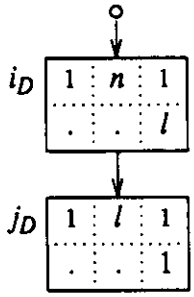
(e) After **&&trans**

Figure 3.6 (cont'd) - Intermediate Code Generation for MM



$$loc(c_{ijk})=ml(i-1)+m(j-1)+(k-1), \quad 1\le i\le n,\ 1\le j\le l,\ 1\le k\le m$$

(f) After &&&*



$$loc(d_{ij})=l(i-1)+(j-1), \quad 1\le i\le n,\ 1\le j\le l$$

(g) After &&(!+) (Final Object)

Figure 3.6 (cont'd) - Intermediate Code Generation for MM

context of functional languages. For example, Pendergrast and Ryder [Pend86] describe a globally optimizing compiler for FP where the following classical optimizations were implemented: local common subexpression elimination with copy propagation; global common subexpression elimination; and global dead store elimination.

The symbolic structural transformations implemented by the compiler described in the previous section also reveal the space-time characteristics of the FP program and this information is used to manage the memory space at run-time. In fact, at the end of compilation we know the amount of memory needed by the program for its entire duration. Using this information, the operating system can emit commands to allocate and deallocate space at run-time. These memory management instructions eliminate the need for garbage collection for the class of programs that can be completely solved structurally.

The same space analysis extracted by the compiler can help in the management of memory hierarchies. Since the compilation treats homogeneous sequences – vectors, matrices, etc. – as contiguous aggregates of data, better use can be made of prepaging schemes and cache memories. This may not be true when homogeneous sequences are treated as general lists, which is the case of the existing interpreters for FP. Contiguous allocation of memory preserves locality when we are dealing with arrays; if arrays are implemented as lists each element can be anywhere in memory.

Our system assumes a linear memory as model. But other models of memory can also benefit from the approach. In the case of interleaved memories, which is a common technique used to increase the bandwidth of memory systems, little modification is needed, since interleaved memories still are linear.

Another aspect where compilation can be of help is in the scheduling of the computation. In the uniprocessor case, the parallel specification given by an FP program must be sequentialized. Note that this process is much easier

than the inverse, where, for example, a FORTRAN sequential specification must be parallelized. But it is in the case of concurrent systems that this problem is more interesting. By compiling the FP program, we can construct the data flow graph of the computation. This graph has nodes representing tasks and arcs representing precedence relationships between tasks. Note that it is much easier to construct such graphs when one is dealing with functional languages, because the precedence relationships can be directly extracted from the text of the program. This is not true for procedural languages: the presence of side-effects entails a complicated analysis to determine the precedence graph. Given the data flow graph, scheduling techniques can be applied to allocate the resources in the concurrent system. Examples of work on scheduling techniques in data-flow graphs for multiprocessors can be found in [Gaud83, Gaud85, Ravi86, Ravi86a].

CHAPTER 4

EVALUATION OF THE COMPILATION APPROACH FOR UNIPROCESSORS

In this chapter we investigate the effects of the structural transformation techniques at compile time on a conventional uniprocessor model. The main motivation for the use of code improving transformation techniques is to relieve the programmer from worrying about performance considerations so that he can concentrate on writing clear code. However, besides preserving the meaning of the programs, the transformations must speed up program execution by a measurable amount. We compare the approach described in the previous chapter with the conventional mode of implementation for FP, namely, interpretation. We show that, for the class of programs characterized in Chapter 2, the structural transformation techniques result in less data replication and less data movement.

4.1 Assumptions and Comparison Measures

We are interested in the comparison between two approaches: the traditional interpreted mode of execution of FP programs versus the compilation mode proposed in this work. We assume that both the compiler and the interpreter run in a conventional von Neumann architecture as depicted in Figure 4.1. The following list describes the assumptions that are made with respect

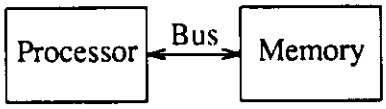


Figure 4.1 - Traditional Uniprocessor

to the machine and the implementation of the interpreter:

a. non-interleaved linear memory;

b. register-to-register architecture, i.e., all machine operations expect operands to be located in processor registers;

c. one-word-at-a-time transfer between processor and memory;

d. one memory word capable of storing any atom (integer, real, character, etc.);

e. string reduction execution mode, which implies that distinct copies of actual objects are generated for each function application;

f. one of the comparison measures that will be made, the memory requirement, defined below, assumes an unlimited amount of memory, that is, we assume that there is no reuse of storage during execution. We want this assumption in order to eliminate the overhead cost of virtual memory management and garbage collection. Actual implementations of FP interpreters must provide some sort of garbage collection that is called upon exhaustion of available space. By assuming unlimited memory, garbage collection becomes unnecessary. Therefore, this assumption places the implementation of the interpreter under a best-

case behavior.

A major motivation in this work has been to minimize the necessity for data movement and data replication in the evaluation of FP programs, and therefore to minimize memory accessing. We will use two comparison measures in order to assess the data movement and data replication that occurs in each approach, compiled and interpreted.

Memory Requirements:

In terms of the uniprocessor model, *data replication* can be measured by the amount of storage for data utilized during the execution lifetime of the program. We will use the amount of data memory required for the execution of an FP program, without reuse of storage, as one of the measures of comparison between the two approaches. Therefore, we define M_{int} as the amount of data memory required for the execution of an FP program in the interpreted case. We denote M^f_{int} for the memory requirements of FP function *f*.

Since the compilation requires the specification of the structure and size of the input object, a fair comparison with an interpreter (which does not require such information) will require a recompilation for each execution of a program. Thus, we divide the memory requirements for the compiled case – M_{comp} – in two parts: M_1 is the memory requirements for the compilation phase and M_2 is the memory requirements for the execution phase. The compilation process has been shown to resemble a symbolic execution of the FP program; as such it utilizes and manipulates descriptors for the input and intermediate objects. Each node of the descriptors occupies 10 words, and the cost

of the compilation phase – M_1 – will be measured in terms of the number of words required for the descriptors.

In order to observe the ratio between M_{int} and M_{comp}, we also define

$$R_M = \frac{M_{int}}{M_{comp}} \tag{4.1}$$

Bus Traffic:

We examine the bus traffic in order to measure the *data movement* required by a program. This can be done by counting the number of loads and stores generated by the program. The rationale is that a fetch requires an item in the memory to be moved from memory to a register in the processor; a store requires a movement in the inverse direction; both require use of the bus. Note that those FP functions that are strictly data movement translate into simple assignment statements of the form A := B or MOVE A,B where MOVE is a memory-to-memory transfer. Since we have assumed a register-to-register architecture, this memory-to-memory transfer translates into a LOAD followed by a STORE. The use of cache memories and/or register files in the processor obviously decreases the bus traffic. However, we do not take into account these effects here, since this decrease would be felt both by the interpretation approach and the compilation approach.

Most, or all, interpreters for FP in uniprocessors use general lists to represent the objects. This means that even regular structures such as vectors and matrices are represented in linked lists. Therefore, data movement for these structures translates into pointer manipulation at the machine level. The

inadequacies of this method of manipulation for regular structures is well-known and has been discussed before in this work as well as elsewhere. To make the comparison easier, we will assume that FP interpreters, upon recognizing a regular object, will represent it using descriptors, thus using index arithmetic instead of pointer manipulation to access its elements. This assumption puts existing interpreters under a better perspective that they really are, and facilitates comparisons because our compiler uses directories to describe and manipulate regular structures.

We denote BT_{int} the bus traffic for the interpreted case and BT_{comp} the bus traffic for the compiled case. Again, we divide BT_{comp} in two components: BT_1 is the bus traffic for the compilation phase and BT_2 is the bus traffic for the execution phase. Finally, we define the ratio R_{BT}:

$$R_{BT} = \frac{BT_{int}}{BT_{comp}} \tag{4.2}$$

In the following sections we examine some characteristic FP programs under the measurements defined above. Section 4.2 examines the well-known Matrix Multiplication program – it is a typical representative of an arithmetic intensive problem. Section 4.3 treats another vector processing problem, but with an additional characteristic: the Fast-Fourier Transform is a recursive program. Section 4.4 looks into purely structural programs; some descriptions of interconnection patterns are studied. Finally, Section 4.5 examines an associative searching problem, which has the characteristic of not being numeric intensive.

4.2 Matrix Multiplication

We now study in detail a comparison between interpretation and compilation for the familiar example of Matrix Multiplication (assume input is of the form <A(n,m), B(m,l)>):

{MM &&(!+) @ &&&* @ &&trans @ &distl @ distr @ [1, trans@2] }

Interpretation Costs for MM

1. Memory Requirements (Table 4.1): Before the beginning of the execution, the data occupies $nm+ml$ words of memory. The first step **[1, trans@2]** requires the replication of the first argument (which requires nm words more), the replication of the second argument (ml words more) and finally the transposition of the second argument (again ml words more). Note that we have to sum up all these contributions, since the string reduction semantics of the interpreter does not allow for reuse of parts of the object. So far, we have used $2nm+3ml$ words of memory.

Step	Words
Initial	$nm+ml$
[1, trans@2]	$nm+2ml$
distr	$nm+nml$
&distl	$2nml$
&&trans	$2nml$
&&&*	nml
&&(!+)	nl
Total	$3nm+3ml+nl+6nml$

Table 4.1 - Memory Requirements, Interpretation of MM

The next step (**distr**) creates a new object that requires $nm+nml$ words. **&distl** creates a new object occupying $2nml$ words of memory. **&&trans** also requires $2nml$ new words. After the execution of all multiplications (**&&&***) the result will require nml words. Finally, the summation reduction (**&&(!+)**) will generate the final answer which requires nl words. Therefore, the direct interpretation of MM requires $3nm+3ml+6nml+nl$ words. For square matrices, for which $n=m=l$, this quantity reduces to

$$M_{inl}^{MM} = 6n^3 + 7n^2 \tag{4.3}$$

2. Bus Traffic (Table 4.2): The first step (**[1, trans@2]**) requires nm fetches and nm stores for the replication of the first argument; ml fetches and ml stores for the replication of the second argument; and ml fetches and ml stores for the transposition. **distr** requires $nm+ml$ fetches and $nm+nml$ stores, which reflects the replication of the elements of matrix B. **&distl** requires $nm+nml$ fetches and $2nml$ stores. **&&trans** requires $2nml$ fetches and $2nml$ stores. The multiplications, **&&&***, require $2nml$ fetches for the operands and nml stores for the results. Finally, **&&(!+)** requires nml fetches and nl stores for the result.

Step	Fetches	Stores	Fetches+Stores
[1, trans@2]	$nm+2ml$	$nm+2ml$	$2nm+4ml$
distr	$nm+ml$	$nm+nml$	$2nm+ml+nml$
&distl	$nm+nml$	$2nml$	$nm+3nml$
&&trans	$2nml$	$2nml$	$4nml$
&&&*	$2nml$	nml	$3nml$
&&(!+)	nml	nl	$nml+nl$
Totals	$3nm+3ml+6nml$	$2nm+2ml+nl+6nml$	$5nm+5ml+nl+12nml$

Table 4.2 - Bus Traffic, Interpretation of MM

We can easily see, by comparing Table 4.1 with the column *Stores* of Table 4.2, that the number of words required is equal to the number of stores issued, except for the initial space required by the input object.

For square matrices ($n=m=l$) the total amount of Fetches+Stores reduces to

$$BT_{int}^{MM} = 12n^3 + 11n^2 \tag{4.4}$$

Compilation Costs for MM

1. Memory requirements for compilation (Table 4.3): The descriptor for the input object <A(n,m), B(m,l)> requires 7 nodes or 70 words. The compilation of the selector 1 generates a descriptor requiring 2 nodes (20 words); the same happens for the compilation of the selector 2. The compilation of **trans** inside the construction requires another 20 words. Table 4.3 lists the requirements for the rest of the compilation. It is important to note that the values in Table 4.3 are the same, independent of the size of the input matrices.

Step	Words
Initial	70
[1, trans@2]	60
distr	60
&distl	60
&&trans	50
&&&*	40
&&(!+)	30
Total	370

Table 4.3 - Memory Requirements, Compilation of MM

Table 4.4 presents the bus traffic costs for the compilation phase of MM.

Step	Fetches	Stores	Fetches+Stores
[1, trans@2]	70	60	130
distr	60	60	120
&distl	60	60	120
&&trans	60	50	110
&&&*	50	40	90
&&(!+)	40	30	70
Totals	340	300	640

Table 4.4 - Bus Traffic, Compilation of MM

Execution Costs for Compiled MM

We recall from Chapter 3 that the pseudo-code for the compiled MM program has four steps: the first and second collapse all data movement that rearranges the matrices in one operation; the third does all multiplications in parallel and the fourth does the summation reductions. As noted in Chapter 3, the pseudo-code is then translated to a real uniprocessor assembler code. Table 4.5 displays the memory requirements for the execution of the compiled program; Table 4.6 displays the results for the bus traffic.

Step	Words
Initial	$nm+ml$
1	$2nml$
2	nml
3	nl
Total	$nm+ml+nl+3nml$

Table 4.5 - Memory Requirements, Execution of Compiled MM

Step	Fetches	Stores	Fetches+Stores
1	$nm+ml$	$2nml$	$nm+ml+2nml$
2	$2nml$	nml	$3nml$
3	nml	nl	$nl+nml$
Totals	$nm+ml+3nml$	$nl+3nml$	$nm+ml+nl+6nml$

Table 4.6 - Bus Traffic, Execution of Compiled MM

For the square matrices case ($n=m=l$), the memory requirements total reduces to

$$M_{comp}^{MM} = 3n^3 + 3n^2 + 370 \tag{4.5}$$

where the constant 370 refers to the cost of compilation. The total bus traffic reduces to

$$BT_{comp}^{MM} = 6n^3 + 3n^2 + 640 \tag{4.6}$$

where, again, the constant 640 refers the the bus traffic during compilation.

Finally, we compare the results:

$$R_M^{MM} = \frac{M_{int}^{MM}}{M_{comp}^{MM}} = \frac{6n^3 + 7n^2}{3n^3 + 3n^2 + 370} \tag{4.7}$$

$$R_{BT}^{MM} = \frac{BT_{int}^{MM}}{BT_{comp}^{MM}} = \frac{12n^3 + 11n^2}{6n^3 + 3n^2 + 640} \tag{4.8}$$

By examining relations (4.7) and (4.8), we can see that, for n very big, the interpretation approach utilizes twice as much memory than the compilation approach; and that the bus traffic in the interpretation case is also twice as bigger than in the compilation case.

By comparing the results for memory requirements and bus traffic obtained in this section, we can see that they differ only by a constant factor, being of the same order of magnitude. More precisely, we can see that the number of stores that contributes to the bus traffic is equal to the memory requirements, except for the initial space required by the input object. This happens because we assumed no reuse of storage and we also assumed that operands for all machine operations must be in processors registers.

4.3 Fast-Fourier Transform

We now examine the performance of the **fftstages** program of Section 2.3.2 [Schl84]. The distinctive property of **fftstages** is that it is recursive and, as such, presents interesting characteristics when compiled. Because of this we want to concentrate only on **fftstages**; we assume that the function **W** is a machine primitive with a fixed execution cost (in fact, the analysis of **W** is trivial and do not affect the results).

First, we note that the function **Bfly** is strictly composed of structural primitives (it is also non-recursive) and, after compilation, it is translated in the following single-step transformation:

- Transfer elements of input object (which is of the form $<a_1, a_2, \cdots, a_N>$, where $N = 2^n$), from positions

$$loc(a_i) = i - 1, \quad 1 \leq i \leq N$$

to positions

$$loc(a_i) = \begin{cases} i-1 & i = 1, 3, \cdots, 2^{n-1}-1 \\ i-2^{n-1} & i = 2^{n-1}+1, 2^{n-1}+3, \cdots, 2^n-1 \\ i+2^{n-1}-2 & i = 2, 4, \cdots, 2^{n-1} \\ i-1 & i = 2^{n-1}+2, 2^{n-1}+4, \cdots, 2^n \end{cases}$$

The cost of compiling **Bfly** in terms of memory requirements is shown on Table 4.7. The results for the bus traffic are in Table 4.8. Note that the cost is the same independent of the number of points in the input object. This is true for any structural-only non-recursive function.

Step	Words
Initial	20
pair	30
split	70
&trans	110
concat	90
[trans@[1,3], trans@[2,4]]	90
concat	70
Total	480

Table 4.7 - Bfly: Memory Requirements for Compilation

Step	Fetches	Stores	Fetches+Stores
pair	20	30	50
split	30	70	100
&trans	70	110	180
concat	110	90	200
[trans@[1,3],trans@[2,4]]	90	90	180
concat	90	70	160
Totals	410	460	870

Table 4.8 - Bfly: Bus Traffic for Compilation

Also note that the condition that ends the recursion of the **fftstages** program (= @ [**length, %2**]) can be solved at compile time, since it is based only on structural information. This enables the compiler to completely unfold the recursion.

The interesting characteristic of the compiled code for **fftstages** is that its size depends on the size of the input object. This is because the compilation, being equivalent to symbolic execution, unfolds the original recursive code. This means, for example, that the size of the compiled code for a 32-point FFT will be greater than the size for a 16-point FFT because the number of recursive calls in the 32-point case is greater. Now we analyze carefully the compilation of **fftstages** in order to clarify its behavior.

Suppose we are compiling a 4-point FFT, which could be considered our basis case. The compilation costs in terms of memory requirements are as follows:

- initially, we need 20 words (2 nodes) to represent the input object;

- the compilation of **Bfly** requires 480 words as shown in Table 4.7;

- we assume that the cost of compiling **&W** is null, since we have assumed that **W** is a machine primitive.

- next, **concat** needs 50 more words; after that we have another **Bfly** that requires another 480 words; the following functions **concat** and **split** require 50 and 70 words more respectively;

- now comes the recursive call to **fftstages**; in this case both halves of the input object to the recursion have length equal to 2; this means that **&fftstages** reduces to **&W**, which has compilation cost zero;

- finally, a **concat** is executed as the last function of the first call to **fftstages** costing 50 words more. Table 4.9 summarizes the counting we have just done. The analysis for bus traffic is similar and the results are shown in Table 4.10.

Step	Words
Initial	20
split@concat@Bfly@ concat@&W@Bfly	1130
&W	0
concat	50
Total	1200

Table 4.9 - fftstages, 4 points: Memory Requirements for Compilation

Step	Fetches	Stores	Fetches+Stores
split@concat@Bfly@ concat@&W@Bfly	20	1130	1150
&W	1130	0	1130
concat	0	50	50
Total	1150	1180	2330

Table 4.10 - fftstages, 4 points: Bus Traffic for Compilation

If we go on to analyze an 8-point FFT, we can observe that its cost can be divided in:

- initial cost: 20 words

- cost of **split@concat@Bfly@concat@&W@Bfly**: 1130 words

- to assess the cost of the recursion **&fftstages**, we observe that each half of the input object is a 4-point object; therefore the cost of this compilation is twice the cost of compiling a 4-point FFT which was shown to be 1200 words.

- finally, the cost of the last **concat** is 50 words.

We can see that, in general, the cost for a N-point FFT ($N=2^n$) is:

$$cost[N\text{-point}] = cost[\text{initial}] +$$
$$cost[\textbf{split@concat@Bfly@concat@\&W@Bfly}] +$$
$$2 \times cost[N/2\text{-point}] +$$
$$cost[\textbf{concat}]$$

or

$$cost[N\text{-point}] = 1200 + 2 \times cost[N/2\text{-point}].$$

The solution for this recurrence is:

$$cost[N\text{-point}] = 1200 \times \sum_{i=0}^{n-2} 2^i, \qquad n \geq 2$$

or

$$M_{\text{fft}} = cost[N\text{-point}] = 1200(2^{n-1}-1), \qquad n \geq 2. \tag{4.9}$$

Similarly, the result for the bus traffic is:

$$BT_{\mathbf{1}}^{fft} = cost[N\text{-point}] = 1180(2^{n-1}-1), \qquad n \geq 2. \tag{4.10}$$

The same line of reasoning can be used to assess the cost of executing a compiled N-point **fftstages**. We begin, again, by analyzing our basis case, the 4-point FFT. Initially, N memory positions are required to store the input object. Since **Bfly** is completely solved at compile time, it will require only N positions more at execution time. Then comes **&W** which requires N new positions for the result. The compilation also collapses in only one step at execution time the composition **split@concat@Bfly@concat**; this step also requires N positions more. Finally, for the 4-point case, **&W** requires N positions more. Note that the final **concat** was solved at compile time and reduces to no operation at execution time. Table 4.11 summarizes the counts for the memory requirements; Table 4.12 presents the results for bus traffic.

Step	Words
Initial	N
Bfly	N
&W	N
split@concat@ Bfly@concat	N
&W	N
Total	$5N$

Table 4.11 - fftstages, 4 points: Memory Requirements for Execution

Step	Fetches	Stores	Fetches+Stores
Bfly	N	N	$2N$
&W	N	N	$2N$
split@concat@ Bfly@concat	N	N	$2N$
&W	N	N	$2N$
Total	$4N$	$4N$	$8N$

Table 4.12 - fftstages, 4 points: Bus Traffic for Execution

We can see now that the memory requirements for a N-point FFT can be described by

$$cost[N\text{-point}] = cost[\text{initial}] +$$

$$cost[\textbf{Bfly}] +$$

$$cost[\textbf{\&W}] +$$

$$cost[\textbf{split@concat@Bfly@concat}] +$$

$$2 \times cost[N/2\text{-point}]$$

or

$$M_2^{fft} = cost[N\text{-point}] = 5N + 2 \times cost[N/2\text{-point}]$$

The solution for this recurrence is:

$$M_2^{fft} = cost[N\text{-point}] = 5N \times \sum_{i=0}^{n-2} 2^i, \qquad n \geq 2$$

or

$$M_2^{fft} = cost[N\text{-point}] = 5N(2^{n-1}-1), \qquad n \geq 2. \tag{4.11}$$

For the bus traffic we have:

$$BT_2^{fft} = cost[N\text{-point}] = 8N(2^{n-1}-1), \qquad n \geq 2. \tag{4.12}$$

To complete the evaluation we study the cost of direct interpretation for the FFT program. Tables 4.13 and 4.14 give the results for the basis case, the 4-point FFT.

Step	Words
Initial	N
Bfly	$7N$
&W	N
split@concat@ Bfly@concat	$10N$
&W	N
concat	N
Total	$21N$

Table 4.13 - fftstages, 4 points: Memory Requirements for Interpretation

Step	Fetches	Stores	Fetches+Stores
Bfly	N	$7N$	$8N$
&W	$7N$	N	$8N$
split@concat@ Bfly@concat	N	$10N$	$11N$
&W	$10N$	N	$11N$
concat	N	N	$2N$
Total	$20N$	$20N$	$40N$

Table 4.14 - fftstages, 4 points: Bus Traffic for Interpretation

The memory requirements for the interpretation of a N-point FFT is given by:

$$M_{int}^{fft} = cost[N\text{-point}] = 21N + 2 \times cost[N/2\text{-point}]$$

which has the following solution:

$$M_{int}^{fft} = cost[N\text{-point}] = 21N(2^{n-1}-1), \qquad n \geq 2. \tag{4.13}$$

The bus traffic for the interpretation is given by:

$$BT_{int}^{fft} = cost[N\text{-point}] = 40N(2^{n-1}-1), \qquad n \geq 2. \tag{4.14}$$

By examining equations (4.9), (4.11) and (4.13), we can see that

$$\lim_{n\to\infty} R_M^{fft} = \lim_{n\to\infty} \frac{21N(2^{n-1}-1)}{1200(2^{n-1}-1) + 5N(2^{n-1}-1)} = 4.2$$

This result tells that the interpretation approach requires 4.2 times more memory than the compilation approach for N very large. Table 4.15 gives the real values for equations (4.9), (4.11) and (4.13) for the first few values of N. We can see that it is best to interpret up to a 64-point FFT; from then on the compilation approach requires less memory.

n	N	Compilation $A=1200(2^{n-1}-1)$	Execution $B=5N(2^{n-1}-1)$	Interpretation $C=21N(2^{n-1}-1)$	$A+B$	$C/(A+B)$
2	4	1200	20	84	1220	0.07
3	8	3600	120	504	3720	0.14
4	16	8400	560	2352	8960	0.26
5	32	18000	1120	10080	19120	0.53
6	64	37200	9920	41664	47120	0.88
7	128	75600	40320	169344	115920	1.46
8	256	152400	162560	682752	314960	2.17
9	512	306000	652800	2741760	958800	2.86
10	1024	613200	2616320	10988544	3229520	3.40

Table 4.15 - fftstages: Comparison for Memory Requirements

Similarly, from equations (4.10), (4.12) and (4.14), we have for the bus traffic:

$$\lim_{n\to\infty} R_{BT}^{fft} = \lim_{n\to\infty} \frac{40N(2^{n-1}-1)}{1180(2^{n-1}-1) + 8N(2^{n-1}-1)} = 5.0$$

4.4 Interconnection Patterns

This section analyzes the bus traffic and the memory requirements for some useful interconnection patterns which have the following FP descriptions:

{shuffle concat @ trans @ split}

{unshuffle concat @ trans @ pair}

{butterfly concat @ concat @ &trans @ &split @ trans @ pair}

{bitreversal (=@[length, %2] -> id;
concat @ trans @ &bitreversal @ split) }

The results are summarized in Tables 4.16 and 4.17.

Program	BT_{int}	BT_{comp}	R_{BT}
shuffle	$6N$	$2N+230$	3.0
unshuffle	$6N$	$2N+210$	3.0
butterfly	$12N$	$2N+770$	6.0
bitreversal	$4N^2-2N$	$2.5N^2-4N$	1.6

Table 4.16 - Bus Traffic for Interconnection Patterns

Program	M_{int}	M_{comp}	R_M
shuffle	$4N$	$2N+140$	2.0
unshuffle	$4N$	$2N+130$	2.0
butterfly	$7N$	$2N+420$	3.5
bitreversal	$2.5N^2-4N$	$2N^2-2N$	1.25

Table 4.17 - Memory Requirements for Interconnection Patterns

In the tables, N is the number of elements of the input object, which must be a power of two. In the last columns we give the limit values of R_M and R_{BT} for N large. The distinctive characteristic of these programs is that they are completely composed of structural primitives, being well-suited for the transformations presented in Chapter 3. The function **bitreversal** is recursive and is analyzed in the same way of the Fast-Fourier Transform of last section. As can be seen by the results in both tables all these functions are significantly improved by the compilation approach in terms of memory usage and traffic in the bus.

4.5 Associative Searching

Now we analyze a problem taken from [Will81]. We study the RANGE program, which is an example of *associative search problem.* Associative searches, or secondary key retrievals, involve a query of some file or data base which is a collection of records or points. Most commonly, a file is organized so that queries based on specific values of a particular key (the primary key) can be processed efficiently. When a query is based on one or more keys other than the primary key, the search process is called *associative*. We assume that the data to be searched comprises n records or points, and that each record has k attributes or keys.

In some of the associative searching problems the query on the file can be answered by combining the answers to the query asked of each subset of an arbitrary partition of the file. For example, the question of whether a particular point is in a file is a decomposable problem. The file can be partitioned in any number of disjoint subsets, each one examined independently to determine if the query point is in that subset, and the results ORed together to produce the result. These problems are called *decomposable*; the RANGE problem examined below falls in this category.

In the RANGE problem, also called "region search", the input consists of n k-dimensional points in a file **F** and a k-dimensional pair of query bounds *(inf, sup)*:

$\mathbf{F} = \{p_1, p_2, \cdots, p_n\}$ where each $p_i = (x_{i1}, x_{i2}, \cdots, x_{ik})$.

query bounds: $\mathbf{Q} = \{q_1, q_2, \cdots, q_k\}$ where $q_j = (inf_j, sup_j)$.

The output is the set of indices of the file points which have all keys greater than or equal to bound *inf* and less than or equal to bound *sup*, that is:

all i such that $inf_1 \leq x_{i1} \leq sup_1$, $inf_2 \leq x_{i2} \leq sup_2$, $\cdots$, $inf_k \leq x_{ik} \leq sup_k$.

Range queries are useful in statistics because they can be employed to determine density estimations, empirical cumulative distributions, and empirical probability contents of hyperrectangles. Range searching is also utilized in geographic data bases. An FP program for RANGE is given in Figure 4.2.

```
# Range Query
# input:  query bounds & n k-dimensional point file
#       : <<query bounds><file points>>
# output: <indices of file points within query bounds>

# Example: Find all records with 1<=key1<=5, 3<=key2<=4, 1<=key3<=4:
# RANGE: <<<1 5><3 4><1 4>><<1 1 1><5 3 4><2 4 3>>> ==> <2 3>

{RANGE   index @ [% T, &rangetest @ distl]}

{rangetest   !and @ &inclusion @ trans}

{inclusion   and @ [<=@1, >=@2] @ distr}

# index: <xi <x1 ... xn>> returns <i1 i2 ... ik> such that
# for all ij, 1<=j<=k  xi = xij
# example: index: <3 <1 2 4 3 3 5 6 7 3 3>> = <4 5 9 10>
#        index: <3 <1 2 4 3 5>> = <4>
#        index: <3 <2 4 6 8 0>> = <>

{index concat @ & (=@2 -> [1]; % <>) @
        trans @ {iota@length@2, distl] }
```

Figure 4.2 - Associative Searching: Range Query

The program first distributes, **distl**, the k range specifications to each file point and then transpose, **trans**, each query/file point pair so that the i^{th} key range and the i^{th} dimensional value are paired. After all this data replication and movement, the inclusion test, **&inclusion**, is applied to all kn pairs to determine which keys lie within the specified range. The result of the test is a sequence of n sequences, each containing k **T** or **F** values. In order to find which points satisfy all k range specifications each sequence is **and**'ed (**!and**), producing n **T** or **F** values. Finally, the function **index** produces the list of indices of those points that satisfy the query. The result is the empty sequence if there are no matches.

Note that the compiler can eliminate a great amount of data replication and movement in the earlier stages of the program. The structural transformations made by the compiler have to stop when the function **index** is encountered, because it has a conditional that depend on the value of the input object. However, by that point, almost the entire program has been already transformed.

The following results are obtained for the memory requirements of the RANGE problem:

$$M_{comp} = 8nk+2k+n+471$$

$$M_{int} = 14nk+2k+n+1$$

$$R_M = \frac{14nk+2k+n+1}{8nk+2k+n+471}$$

Since in general $k \ll n$, R_M approaches 1.75 for large values of n. This means that the interpreted execution of RANGE uses 75% more memory than the compiled version.

For the bus traffic, we have the following results:

$$BT_{comp} = 15nk+2k+n+831$$

$$BT_{int} = 27nk+2k+n+1$$

$$R_{BT} = \frac{27nk+2k+n+1}{15nk+2k+n+831}$$

For $k \ll n$, R_{BT} approaches 1.80 for large values of n, which means that the interpreted execution of RANGE has 80% more bus traffic than the compiled version.

4.6 Conclusion

In this chapter we have analyzed the performance of a number of representative FP programs with respect to their usage of memory and bus traffic under the compilation techniques presented in Chapter 3. Analytical results derived from a simple model of uniprocessor show that the structural transformations done by the compiler do indeed decrease data replication and data movement in the execution of FP programs.

Table 4.18 summarizes the memory requirements improvements for the FP programs analyzed in this chapter. In the table, R_M shows the limit value of R_M for very large input objects.

Program	R_M
MM	2.0
fftstages	4.2
shuffle	2.0
unshuffle	2.0
butterfly	3.5
bitreversal	1.25
RANGE	1.75

Table 4.18 - Summary of Memory Requirements Results

Finally, the summary of results for the bus traffic is shown in Table 4.19 – R_{BT} shows the limit value of R_{BT} for very large input objects.

Program	R_{BT}
MM	2.0
fftstages	5.0
shuffle	3.0
unshuffle	3.0
butterfly	6.0
bitreversal	1.6
RANGE	1.8

Table 4.19 - Summary of Bus Traffic Results

CHAPTER 5
ISSUES IN PIPELINED AND MULTIPROCESSOR SYSTEMS

This chapter discusses compilation issues for pipelined and multiprocessor systems. We show that the ideas developed in the previous chapters are also useful when the machine model is not an uniprocessor. The first section deals with vector processing and pipelined computers. These still are von Neumann architectures, but incorporating improved techniques to speed up the execution of arithmetic-intensive, vector processing problems. Then, we discuss the use of compilation in the control of interconnection networks, a crucial element for multiprocessor architectures. Finally, a string-reduction architecture developed for the specific purpose of directly interpreting FP programs is examined; we show that this model also can take advantage of the compilation techniques.

5.1 Vector Processing in FP

Vector processing machines – like the Cray-1 [Cray76] and the CYBER-205 [Kasc79] – were specially designed to work more efficiently with one-dimensional lists of data (vectors). These machines have been commercially available for a decade now, and apart from some good vectorizing compilers [Arno82], we have yet to see significant improvements in software development systems for them. As pointed out in Chapter 1, these machines

run FORTRAN-coded programs – in order to obtain a significant speed-up either the FORTRAN program must be compiled by a vectorizing compiler or the user must hand-code (in assembler) the critical points of the program.

In this section, we try to show that functional languages can be an useful tool to specify vector problems. We will see that the parallelism of vector problems can be easily expressed in FP and the compilation techniques presented in Chapter 3 can be used to generate efficient code for pipelined machines.

We first observe that, in these machines, the operand specification part of the instruction repertoire normally have several memory address fields: the origin of the vector, the dimensionality; the number of elements per dimension; the data type of each element; the arrangement of elements in memory. We recall from Chapter 3 that the object descriptors keep all this information and therefore the full capability of the machine instructions can be exploited.

Vector processing applications have been historically written in FORTRAN. However, it is clear that this language is not a good one to specify parallel problems. Apart from the worst characteristic of being a sequential language, FORTRAN presents some implementation properties that further hinder the exploitation of parallelism. For example, FORTRAN has an implicit ordering, column-wise, for the storing of matrices. It is not a good practice to encode an ordering in the program. If no ordering is encoded the compiler may choose the most efficient ordering for the target computer. Moreover, should the target computer contain parallelism, then some or all of the operations may be performed concurrently.

Below a number of common vector processing operations are presented and described in FP. Then we point out how the transformation approach described in Chapter 3 is specially well-suited for vector processing. We also present the FORTRAN code of some of the operations with the intent of exposing the different approaches represented by the FORTRAN style and the FP style of programming.

Unary Vector Operations:

$\&f$: X(1:n) $\equiv \langle f:x_1, \ldots, f:x_n \rangle$,

where f is any unary operator (function).

Binary Vector Operations:

$\&f$@trans: <X(1:n), Y(1:n)> $\equiv \langle f:\langle x_1, y_1 \rangle, \cdots, f:\langle x_n, y_n \rangle \rangle$,

where f is any binary operator (function). Binary operations can be extended as follows to higher-ranking objects (arrays). For two-dimensional matrices the FP description is:

$\&\&f$ @ &trans @ trans

In general, for a k-dimensional array, we have:

$\&^k f$ @ $\&^{k-1}$trans @ $\&^{k-2}$trans @ $\cdots$ @ &trans@trans

This pattern can be used also to implement common vector machine operations such as "vector compare on equal" – VCOMPEQ (X, Y, Z):

$$Z(I) = \begin{cases} 1 & \text{if } X(I) = Y(I) \\ 0 & \text{otherwise} \end{cases}$$

The FP implementation of this operation is simply **&eq @ trans.**

Recall from the previous chapter that M_{int} is the amount of data memory required for the execution of the FP program in the interpreted case; M_1 is the memory requirement for the compilation phase; M_2 is the memory requirement for the execution phase of the compiled code; M_{comp} is the sum of M_1 and M_2 and $R_M = M_{int}/M_{comp}$.

The memory requirements of binary array operations is given by the following equations, where k is the dimensionality of the array and n is the number of elements along each dimension (we assume that each dimension has the same number of elements because practical problems generally are in accordance with this assumption).

$$M_1 = 2k^2 + 5k + 7$$

$$M_2 = 5n^k$$

$$M_{int} = (2k + 2)n^k$$

$$R_M = \frac{(2k + 2)n^k}{5n^k + 2k^2 + 5k + 7}$$

Note that, as expected, the memory requirements for the compiler (M_1) depend only on the dimensionality of the input arrays and not on their size. Since $k \ll n$, we can see that $R_M = O(k)$. Therefore, no real gains are obtained for vectors (k=1), operations on matrices utilize twice as much memory when

interpreted and so on.

Selection from Array Objects

We distinguish four cases, following [Hock81]:

1. *Selecting reduced rank objects*: This corresponds to indexing in a sequential language, with the difference that lower-rank objects may be selected. For example, in a matrix we may want to select the whole matrix, a column, a row or a single element. In FP, this selection is straightforward and is mostly done by using selectors. Thus, given a two-dimensional $N \times M$ array A, we may wish to select:

a. the whole matrix: **id**

b. the i^{th} row: selector **i**

c. the j^{th} column: **j @ trans** or **&j**

d. the i, j element of the matrix: **j @ i**

2. *Selecting range of values*: Necessary to select sub-matrices or to reduce the size of objects. For example, given an $N \times N$ matrix A, we may wish to select the interior points of the matrix, i.e., matrix A(2:N-1, 2:N-1). The following FP program does the job for a 5×5 matrix, using selectors only:

```
{range      [[2,3,4] @ 2,
             [2,3,4] @ 3,
             [2,3,4] @ 4 ]}
```

Although very straightforward, this function generates excessive copying if

interpreted under a string-reduction semantics – the matrix would have to be copied three times and then each selected row would be copied three times. By compiling first, all this copying is avoided. The same problem can be solved more efficiently by the following FP program, which does not rely on the exclusive use of selectors:

{range &tlr @ &tl @ tlr @ tl}

Note that this function is more in the spirit of FP, since it selects the interior points of matrices of *any* size, and not only 5×5 as the previous program. This function is more efficient, in the interpretation case, because only one copy of the original matrix is necessary; subsequently, reduced-range copies are passed to each step of the composition. This function is also more efficient in the compilation case, since it generates a smaller descriptor for the result object. The fundamental difference between the two versions is that the first one generates 9 distinct sub-objects (**2@2, 2@3, 2@4, 2@3**, etc.), while the second generates only one sub-object at each step of the composition. It can be argued that the first version has more parallelism, but the parallelism is gained at the expense of much more space. Also, the parallelism argument loses strength if the program is compiled, because both versions are collapsed into a single step.

Any arbitrary combination of range-reducing selections can be implemented in FP by using a combination of selectors and/or structural functions such as **tl** and **tlr**. In any case, an interpreter will have to make several copies of parts of the input array, whereas the compiler avoids all copying.

3. *Selection using indirect addressing*: The implementation of indirect addressing in FP generally requires the use of the primitive **pick**, which cannot be solved at compile time since it is data-dependent. As an example we show the FP implementation of the *gather* instruction, which is one of the machine instructions of the CDC Cyber 205. Gather can be described by the following FORTRAN loop:

```
   DO 1 I=1, N
1  Y(I) = X (INDEX(I))
```

One possible FP implementation is shown in Figure 5.1. Note the recursive structure and the presence of the primitive **pick**.

```
input:   <<index vector><data vector>>
output:  <permutation of data vector cf. index vector>
we assume index vector is a permutation of (1:N)

E.g.: gather: <<2 4 6 1 3 5><15 10 30 17 12 11>> = <10 17 11 15 30 12 >

{setup [1, [], 2]}   # setup input obj: < <indices> <> <data> >

{f    [tl@1,
       apndr @ [2, pick @ [1@1, 3]],
       3] }

{gat    (null@1 -> 2; gat @ f) }

{gather   gat @ setup}
```

Figure 5.1 - FP Implementation of `gather` Instruction

Note that compilation avoids copying in the first step, setup. However the above function cannot be completely solved by the methods described in Chapter 3. But compilation still offers the opportunity of identifying the pattern as a machine instruction, therefore generating efficient run-time code.

Some work on this area of pattern-matching has been referenced in Chapter 1.

4. *Selection using boolean vectors*: This operation, called *compress*, is found as a primitive in APL and is also a machine instruction in the Cyber 205. Given a bit vector and a data vector it compresses the data vector for corresponding entries in the bit vector that are equal to 1. One FP implementation in shown in Figure 5.2. Again, the conditional is data-dependent and not solvable by the methods of Chapter 3, but pattern matching techniques are also applicable here.

```
input: <<control bits vector><data vector>>
if bit-i is 1 then select data-i else don't select
Example: compress: <<1 0 0 1 1 0 1><1 2 3 4 5 6 7>> = <1 4 5 7>

{compress    concat @ &(eq@[1,%1] -> [2]; %<>) @ trans}
```

Figure 5.2 - FP Implementation of `compress` Instruction

Linear Combination of Vectors

input: << a, X(1:n) > < b, Y(1:n) >>

output: $<ax_1+by_1, ax_2+by_2, \cdots, ax_n+by_n>$

where multiplication and addition stand for any binary operators:

&+ @ trans @ &&* @ &distl

Vectorsums

input: X(1:n)

output: $x_1 + x_2 + \cdots + x_n$,

where addition also stands for functions such multiplication, max, min, etc. This can be directly implemented by the insert functional form of FP.

Vector Innerproduct and Matrix Multiplication

We have already studied matrix multiplication (MM); inner product is part of MM:

{IP (|+) @ &* @ trans }

Although the above list of common vector operations is quite representative, real life problems often present a varied mix of these patterns. To investigate this aspect, we present FP implementations of some Lawrence Livermore Loops in the next section.

Lawrence Livermore Kernels

The Livermore Loops Benchmarks have been used for over a decade to gauge the performance of supercomputers (for example, [Riga84]) and of FORTRAN optimizing compilers for supercomputers [Arno82]. They comprise a series of FORTRAN kernels extracted from representative programs that are used in various U.S. national laboratories. In the following, we use the FORTRAN description of the kernels as they appear in [Riga84]. We develop implementations of some of the kernels in FP. In order to better characterize the kernels, we divided them in groups.

Group 1: Kernels 2 and 3

Both kernels are inner products; in kernel 2 it is camouflaged by being unrolled into a sum of five partial products for each iteration of the loop. The previous section showed the FP implementation for inner product.

Group 2: Kernels 1, 7, 9, 10, 12

We present the FP description of kernel 7 in Figure 5.3. The other kernels can be found in Appendix 3. The distinctive characteristic of these kernels is that all iterations can be done in parallel. As can be seen by the FORTRAN description, this may not be clear, at least for some of them. Also, not all compilers studied in [Arno82] are capable of uncovering the hidden parallelism of some of the kernels. A characteristic of the FP implementations is a first step that aligns the operands for the subsequent operations. This alignment step is purely structural and requires substantial copying in an interpretive implementation of FP.

All the FORTRAN vectorizers examined in [Arno82] can vectorize the loop of Figure 5.3 – albeit using the complicated analysis that is required in such cases; on the other hand, the FP program is already in a form that exposes the maximum parallelism. Also, note that the FP program can be applied to any vector size (greater than 7), while the FORTRAN program is bound to vectors of size 120.

Kernel 7: Equation of State Excerpt

FORTRAN:

```
DO 7 M = 1, 120
 X(M) = U(M) + R* (Z(M)+R*Y(M))+
    + S * ( U(M+3)+R*(U(M+2)+R*U(M+1))
7           + S * (U(M+6) + R*(U(M+5)+R*U(M+4))))
```

FP: input object: < Y(1:120), Z(1:120), U(1:126) >
output object: X(1:120)

```
{Ym 1}   # select Y(1:120)
{Zm 2}   # select Z(1:120)
{Um        tlr @ tlr @ tlr @ tlr @ tlr @ tlr @ 3 }   # select U(1:120)
{Umplus1   tlr @ tlr @ tlr @ tlr @ tlr @ tl  @ 3 }   # select U(2:121)
{Umplus2   tlr @ tlr @ tlr @ tlr @ tl  @ tl  @ 3 }   # select U(3:122)
{Umplus3   tlr @ tlr @ tlr @ tl  @ tl  @ tl  @ 3 }   # select U(4:123)
{Umplus4   tlr @ tlr @ tl  @ tl  @ tl  @ tl  @ 3 }   # select U(5:124)
{Umplus5   tlr @ tl  @ tl  @ tl  @ tl  @ tl  @ 3 }   # select U(6:125)
{Umplus6   tl  @ tl  @ tl  @ tl  @ tl  @ tl  @ 3 }   # select U(7:126)

# align operands
{align  [ Ym, Zm, Um, Umplus1, Umplus2, Umplus3, Umplus4, Umplus5, Umplus6 ] }

#  Expression will be divided in three clusters -- c1, c2, c3 --

{c3   * @ [S, + @ [9, * @ [R, + @ [8, * @ [R, 7 ] ] ] ] ] }

{c2   + @ [6, * @ [R, + @ [5, * @ [R, 4 ] ] ] ] }

{c1   + @ [3, * @ [R, + @ [2, * @ [R, 1 ] ] ] ] }

{Xm   + @ [c1, * @ [S, + @ [c2, c3]]] }   # the whole expression is c1 + S * (c2 + c3)

# LOOP:

{LLL7   & Xm   @   # execute all expressions
        trans  @   # put elements for each expression together
        align  }   # create and align all copies
```

Figure 5.3 - Livermore Kernel No. 7

Group 3: Kernels 5, 6, 11

These kernels are examples of first-order linear recurrences of the form:

$$x_j = a_j x_{j-1} + d_j, \qquad 1 \le j \le n \tag{5.1}$$

This well-known formulation seems to be inherently sequential; however, parallel formulations exist (for example, [Kuck78, Hock81]). First-order linear recurrences occur very frequently in numerical calculations: solution of linear equations by Gaussian methods, iterative methods in which a better approximation to a solution is calculated from previous approximations, solution of ordinary differential equations.

There is an elegant way to implement this type of recurrence in FP using tail-recursion as shown bellow for an input of the form < x_0, A(1:n), D(1:n) >:

```
{f      (null@2 -> 1; f @ [apndr@[1, expression], tl@2, tl@3]) }

{expression + @ [ *@[1@2, last@1], 1@3]}
```

This implementation is done using the following general pattern:

```
{f   (p -> g;  h @ f @ k)  }
```

For this particular case, function h is absent, or could be considered to be the FP function **id**. The implementation of Kernel 6, shown in Figure 5.4 (Kernels 5 and 11 are in Appendix 3), was also done using this pattern.

Kernel 6: Tri-diagonal elimination, above diagonal

FORTRAN:

```
DO 6 J = 3, 999, 3
 I = 1000 - J + 3
 X(I) =  X(I) - Z(I)*X(I+1)
 X(I-1) = X(I-1) - Z(I-1)*X(I)
 X(I-2) = X(I-2) - Z(I-2)*X(I-1)
6 CONTINUE
```

This loop can be rewritten as:

```
DO 6 J = 1, 999, 1
  I = 1000 - J + 1
  X(I) = X(I) - Z(I)*X(I+1)
6 CONTINUE
```

FP: input object: < X(1:1001) Z(1:1000) >
output object: X(1:1000)

```
{setup   [tl@tlr@1, [last@1], tl@2]}

{recur (null@1 -> 2;
                  recur @ [tlr@1, apndl@[expression, 2], tlr@3])
}

{expression   - @ [last@1, * @ [1@2, 1@3]] }

{LLL6 recur @ setup}
```

Figure 5.4 - Livermore Kernel No. 6

This form has been considered by Backus as an important one and is treated in his original article in the Iteration Theorem. There, he shows a simple expansion for functions that follow the above pattern. The work by Kieburtz and Shultis [Kieb81] also deals with this type of recursive functions. They present a scheme for high-level program transformation based on pattern matching which is directed towards transformation of tail-recursive functions

into iterative versions. They argue that execution of recursive functions is inefficient in conventional processors and thus should be transformed to iterative form. In particular, they show that the pattern

{*f* (p -> g; f @ k) }

can be evaluated iteratively as

{*f'* g @ (while not @ p k) }.

The same technique of pattern matching can be used to identify these instances of first-order linear recurrences, and be applied to generate efficient vector code for pipelined von Neumann machines. For example, in [Arno82] the three kernels 5, 6 and 11 are left as scalar code by two of the optimizing compilers studied in the article. The other vectorizing compiler, KAP [Wolf81], considered to be the state-of-the-art in compiler technology for supercomputers, is able to recognize the first-order linear recurrence and to generate a macro instruction, for which the CYBER 200 FORTRAN generates a STACKLIB call, which implements such recurrences using the vector capabilities of the machine. The point here is that it is not trivial (but it is possible, as in the KAP case) to recognize this common pattern in languages such as FORTRAN; on the other hand a simpler pattern matching mechanism is able to identify the recurrence in the FP case. The compiler described in Chapter 3 does not implement any kind of pattern recognition, but compilation opens the possibility of implementing these transformations and it can be incorporated in FP systems as is shown in [Kieb81].

On the other hand, the recursion step of these linear recurrences is terminated by a structural predicate, which can be solved at compile time since it is in accordance with the restriction defined in Chapter 2 for the conditional functional form.

Livermore Kernels Performance

Table 5.1 presents the memory requirements for the compilation and interpretation of the Livermore Kernels discussed above. In the table, n is the size of the input vectors; the column R_M shows the limit value of R_M for very large values of n; and the column n_{cross} shows the smallest size of the input object for which the value of M_{comp} turns smaller than the value of M_{int}.

Kernel	M_{int}	M_{comp}	R_M	n_{cross}
1	$27n+143$	$5n+901$	5.40	35
2,3	$3n+1$	$3n+81$	1.00	∞
5	$4n^2+5n-3$	$1.5n^2+176n+298$	2.67	71
6	$4n^2+2n$	$1.5n^2+175n+210$	2.67	71
7	$53n+153$	$12n+1406$	4.42	31
9	$160n+28$	$53n+934$	3.02	9
10	$54n$	$22n+200$	2.45	7
11	$3n^2-n+2$	$n^2+142n-50$	3.00	72
12	$5n$	$3n+110$	1.67	56

Table 5.1 - Memory Requirements for Livermore Kernels

Table 5.2 presents the results for the bus traffic. As in Table 5.1, n is the size of the input vectors; R_{BT} shows the limit value of R_{BT} for very large values of n; and n_{cross} shows the smallest size of the input object for which the value of BT_{comp} turns smaller than the value of BT_{int}.

Kernel	BT_{int}	BT_{comp}	R_{BT}	n_{cross}
1	$50n+275$	$5n+1671$	10.00	32
2,3	$6n+3$	$6n+163$	1.00	∞
5	$8n^2+7n-7$	$3n^2+348n+505$	2.67	70
6	$8n^2+n$	$3n^2+344n+350$	2.67	70
7	$98n+294$	$12n+2546$	8.17	27
9	$306n+49$	$92n+1781$	3.33	9
10	$86n$	$22n+310$	3.91	5
11	$6n^2-6n+4$	$2n^2+282n-140$	3.00	72
12	$7n$	$3n+160$	2.33	40

Table 5.2 - Bus Traffic for Livermore Kernels

5.2 Control of Interconnection Networks

Concurrent processing depends on interconnection networks for communication among processors and memory modules. Today, with the advent of VLSI technology, it is possible to construct a concurrent processing system by interconnecting hundreds of off-the-shelf processors and memory modules. For our purposes, assume a parallel processing machine without shared memory. Also, assume that the processing elements (PE's) of this machine are connected through a switching network as in Figure 5.5.

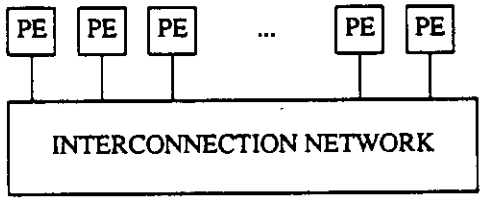


Figure 5.5 - Hardware Model of Parallel Processing Systems

A *generalized connection network* (GCN) is a switching network with N inputs and N outputs that can be set to pass any of the N^N mappings of inputs to outputs. If we restrict the mappings only to one-to-one mappings, then $N!$ such mappings are well-defined. A network that performs these $N!$ mappings is called a *permutation network.*

Parker [Park80] showed that 3 passes through some shuffle/exchange-type permutation networks are sufficient to generate any permutation, 2 passes being necessary. Each pass through these networks takes $O(logN)$ time. The proper switch settings for any desired permutation pattern may be found in $O(NlogN)$ time [Waks68] on a serial computer. In 1978, Thompson [Thom78] defined a GCN that uses less than $7.6NlogN$ contact pairs and has $O(logN)$ delay. The algorithm to set up the switches of this GCN also takes $O(NlogN)$ time on a serial computer.

One step in the problem of finding the setting of the switches of a switching network can be formalized as a sequence of N integers, one for each output vertex: $j_1, j_2, \cdots, j_N$ where each $j_k = i$ iff output number k is connected to input number i (each output is connected to exactly one input). For example, if $N = 4$ a setting might be (3, 3, 4, 1): input 3 is connected to outputs 1 and 2, input 4 to output 3, and input 1 to output 4.

Using the transformation techniques of Chapter 3 more effective use of interconnection networks can be achieved. This is because the sequence of integers that characterize the switch setting of a network can be obtained directly from the object descriptors of the compiler. Given the sequence, it is necessary to calculate the switches settings; this can be done, statically, at

compile time. Also, sometimes it is better to have equations describing the input-output mapping of the network, since a set of equations can better reveal the structure of the interconnection pattern. Again, the set of equations can be directly extracted from the object descriptors and the switch settings can be calculated at compile time. Of course, the compile time will increase by an amount proportional to the complexity of calculating the switch settings; however, no control penalty is paid at run time.

As an example, consider the function **butterfly** that was presented in Section 4.4 and which is also part of the Fast Fourier function of Section 4.3:

{butterfly concat @ concat @ &trans @ &split @ trans @ pair}

The input object, which has the form $<a_1, a_2, \cdots, a_N>$, where $N = 2^n$, has the following equation to represent the positions of its elements:

$$loc(a_i) = i-1, \quad 1 \le i \le N \tag{5.2}$$

After compilation, the final positions of the elements are represented by the following set of equations:

$$loc(a_i) = \begin{cases} i-1 & i = 1, 3, \cdots, 2^{n-1}-1 \\ i-2^{n-1} & i = 2^{n-1}+1, 2^{n-1}+3, \cdots, 2^n-1 \\ i+2^{n-1}-2 & i = 2, 4, \cdots, 2^{n-1} \\ i-1 & i = 2^{n-1}+2, 2^{n-1}+4, \cdots, 2^n \end{cases} \tag{5.3}$$

For $N = 16$, Equation 5.2 represents the following sequence:

{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15}

and Equation 5.3 stands for the sequence bellow:

{0, 8, 2, 10, 4, 12, 6, 14, 1, 9, 3, 11, 5, 13, 7, 15}

With the knowledge of these sequences at compile time, the switch settings of the particular network utilized by the parallel processing system can be calculated at compile time, saving time when the program is executed.

It is important to notice that a parallel system that incorporates a network but directly interprets FP programs cannot utilize the full potential of the network because it will use the network only to implement FP primitives that require data movement: **trans, distl, distr**, etc. As a consequence, only simple networks that are capable to realize the FP primitives are necessary at machine level, resulting in not very powerful machines. The problem is that there is no point in using more powerful networks to implement only simple FP primitives; they would be under-utilized.

5.3 String Reduction Machines

In this section we point out how compilation of FP can be of use to novel proposed architectures. We focus attention on the general class of string reduction machines, specifically on the one proposed by G. Magó [Mago79]. In string reduction machines instructions are executed by a substitution process, which locates reducible applications in the program text and replace them by other expressions that have the same meaning, until a constant expression representing the final object is reached.

Magó's machine was specifically designed to execute FP. It considers FP as its machine language and automatically exploits the parallelism present in FP programs. The machine is a *small-grain* multiprocessor consisting of a large number of "cells". It has the overall structure shown before in Figure 5.5. The PE's in Magó's machine are called *L-cells* (for "leaf" cell) and the interconnection network is a full binary tree, each node of the tree being named a *T-cell* (for "tree" cell). In addition to the tree network, each L-cell is connected with its two neighbors to form a linear array.

An FP machine program is a linear string of symbols that are mapped into the vector of L-cells so that each L-cell holds one symbol of the FP program. Some syntactical separators of FP are omitted, since cell boundaries fulfill their purpose. Also, sequencing brackets are substituted by integers that indicate the nesting level of the sequence.

The L-cells are microprogrammed to identify the reducible applications (RA's) and build sub-trees, linked by the T-cells, that become dedicated to the reduction of that RA. The reduction of each application is handled by microprograms that normally reside outside the machine and are brought on demand. Once a microprogram is loaded in the registers of the L-cells, each L-cell executes the necessary steps to make its contribution to the total reduction.

An important phase in each execution cycle of the machine is called *storage management*. Storage management is necessary during computation because FP programs can expand and shrink while being executed. For example, the primitive **distl** broadcasts one of its operands to the other and thus

needs more space than the original object. If there is not sufficient space in adjacent cells, storage management must rearrange the program in order to provide the needed cells.

It turns out that the acquisition of space during storage management can be extremely expensive in Magó's machine. This problem has received considerable attention by Magó and co-workers as can be seen in [Mago84, Will81, Stan81, Mago81]. For example, the article [Mago84] analyzes the performance of some matrix algorithms on Magó's machine. Performance is predicted in terms of execution time of programs. The model accounts for communication costs and storage management costs. Since the cost of storage management depends on the initial layout of the program text on the L-cells a lower bound and an upper bound are obtained. The lower bound assumes a lucky initial layout of the program, i.e., whenever k symbols are to be inserted between two adjacent symbols of the initial program text, these two symbols are situated in L-cells with at least k empty cells in between. The upper bound assumes that the initial program text is fully compressed, and that the application is situated at one end of the L-array.

Five different implementations of the matrix multiplication program (MM) are analyzed under the assumptions. MM1 is our familiar MM presented in Section 2.3.1. MM2 is a variant of MM1 with a new functional form "apply-to-the-rightmost-element" (**AR**) so that **[1, trans@2]** can be substituted by **AR trans**. As mentioned in Chapter 1, this new form avoids unnecessary copying, lessening the execution time. Both MM1 and MM2 (which are fully parallel), have $O(n^3)$ for upper bound and $O(n^2)$ for lower

bound in execution time. This shows that their performance is heavily dependent on the initial layout of the program text. The reason is that to acquire $O(n^3)$ storage cells in Magó's machine, $O(n^3)$ time is required.

MM3 is an intermediate program between fully parallel and fully sequential. It requires $O(n^2)$ space and has upper bound of $O(n^3)$ and lower bound of $O(n^2)$ as MM1 and MM2. However, the coefficients for the polynomials are much lower. MM4 is a variant of MM3 with more powerful primitives, specially microprogrammed for this problem with both an upper bound and lower bound of $O(n^2)$. Finally, MM5 is a single machine primitive microprogrammed to compute the product of the two matrices. This microprogram utilizes temporary registers of the L-cells and the sequencing capability of the microprogramming language. Both the upper bound and the lower bound are $O(n^2)$ with lower coefficients than MM4. This bound seems to be the best one can achieve using a tree network.

We can see that the difference between the upper bound and the lower bound can be slight (as in MM5) or considerable (as in MM1 and MM2). This indicates how sensitive execution time is to the initial distribution of the FP symbols in the L-array. The authors conclude by saying that indiscriminate copying (to gain maximum parallelism) can hurt the execution time and that the aims of optimizing an implementation should be to minimize data movement and storage management.

Magó, in [Mago84a], argues that the efficient use of language-based parallel computers is a *programming problem* and that either the *user* or an *optimizer* must be responsible for constructing efficient programs. Both would

rely on high-level guidelines like "reduce copying" and "reduce storage management" for improving the execution time. However, if the user has to worry about this kind of guidelines he must know about machine implementation details, which may undermine the motivation to use very high-level languages in the first place. No user is expected to be worried with the initial layout of the program in the machine when developing the algorithmic solution for a problem. Furthermore, if the user has a language in which he can specify the maximum parallelism that the problem presents, which is the case of FP, we cannot expect him to try to decrease the parallelism of the implementation because too much parallelism may hurt.

On the other hand, the *optimizer* suggested by Magó is a worthwhile goal to strive for. Such an optimizer for Magó's machine can incorporate many of the ideas developed in this work, as discussed below.

First, when an FP program is compiled as in Chapter 3, the memory requirements for the execution will be known beforehand. This information can be used to check whether the machine has sufficient L-cells to execute the entire program or not. If not, the compiler can issue directions for scheduling the execution of the program. Conversely, if the machine has L-cells in abundance, the compiler can use the extra space to find a good initial layout for the program. In fact, this problem is treated by Stanat and Magó [Stan81] for the problem of finding the extra space during execution time. The same procedure can be used to determine statically, at compile time, the initial layout of the program.

Second, there is the problem of the utilization of the network. Note that the tree network of Magó's machine implements only the communication patterns present in the FP primitives. If one needs more sophisticated patterns of data movement, one must re-microprogram the machine as was done in the MM5 implementation of the matrix multiplication program. At this point, the question arises why the network in Magó's machine is a tree network. One early justification is that reducible applications in FP programs present a high degree of locality and therefore a tree network is a good candidate. But the fact that a tree is not sufficiently powerful was realized by Kellman [Kell83] who presented a proposal to incorporate parallel sorting networks, like the bitonic sorter of Batcher [Batc68] in a machine architecture similar to Magó's machine. Kellman showed that indeed a more powerful network speeds up the execution of programs. More recently, Plaisted [Plai85] refined the ideas of Kellman and proposed a shuffle/exchange network for Magó's machine. In Plaisted's work we note that, in order to fully utilize the power of the network, one must depart from simple FP primitives and microprogram the machine to realize the more elaborated data movement patterns. He then shows a microprogrammed implementation of the matrix multiplication program that runs in $O(log^2 n)$, in contrast with the $O(n^2)$ obtained by the original machine with the tree network. Here, we return to the aspect discussed in the last section: there is no point in providing a powerful network if the machine is going to interpret the basic FP primitives only. Again, the transformation techniques presented in this work are of help here since the algebraic equations generated by the compiler can be used to calculate the switch settings of the network at compile time, thus exploiting the full capabilities of the network.

CHAPTER 6

CONCLUSIONS AND SUGGESTIONS FOR FUTURE RESEARCH

6.1 Conclusions

The applicative style of programming presents an alternative paradigm for the specification of algorithms. Functional languages provide the programmer with a powerful high-level tool for the design of programs. They present many advantages such as referential transparency, freedom from side-effects, expandability and modularity. Also, the cleaner and concise semantics, based on the mathematical concept of functions, allows for an easier verification of the correctness of functional programs. Finally, implicit and explicit parallelism are easily representable in functional languages. All these reasons make functional languages a prime candidate for the basis of future generation computer systems.

However, there are sources of inefficiency in the execution of functional programs that are likely to interfere in performance irrespective of machine implementation. These sources of inefficiency are related to excessive data movement and excessive data copying that can occur in functional programs as a result of the programming style. The dissertation began by identifying these problems and then we worked on a new approach to improve the efficiency in the execution of functional programs.

The dissertation showed how structural transformation techniques can be applied at compile time in the context of functional programs. Although functional language systems have traditionally been interpretive, we see no reason why compilation cannot be applied to them. The freedom from side-effects presented by functional languages is a big asset for the compiler designer, resulting in a much simpler data-flow analysis of programs as compared with the case of imperative programs. Furthermore, the powerful mathematical properties of functional languages can be used – as they were in this work – to further increase the scope of compilation.

The algebra of the language was used to develop an algebra of structural computations. This algebra is then used as the basis for the development of the compiler. In the compilation process, algebraic equations are used to represent the structure and the location of FP objects in a given memory organization. The manipulation of these algebraic equations allows the solution of some FP primitives at compile time; this process minimizes the amount of data replication and data movement required by the original FP program. The specific result is a more efficient run-time environment with respect to a given architecture for FP programs.

An analysis of the compiler on a variety of programs showed that indeed savings in data movement and data replication are obtained when FP programs are compiled. The analysis also showed that the class of problems that benefits most from the compilation approach is composed of programs that manipulate regular data structures such as vectors and matrices.

The structural transformation techniques were also shown to be useful when the machine model is not an uniprocessor. Some vector operations as well as some actual vector problems were implemented in FP and the analysis of their performance showed an improvement when the structural transformations were applied. Another aspect of multiprocessor machines that also can take advantage of the techniques is the control of interconnection networks. The algebraic equations that describe the positions of the FP objects can be utilized as input specification for algorithms that calculate the switch settings of these networks, thereby solving this problem at compile time. Finally, Magó's machine, a string-reduction architecture developed for the specific purpose of executing FP programs, was examined, and we indicated how it can take advantage of the structural transformation techniques.

In summary, this work has shown:

1. the development of an algebra of structural transformations for FP programs that is used as a basis for the implementation of compiler techniques for the an FP system;

2. the description of a compiler for FP that utilizes algebraic equations to represent the structure and the location of FP objects in a given memory organization. The manipulation of these algebraic equations allows the solution of some FP primitives at compile time; this process minimizes the amount of data replication and data movement required by the original FP program, resulting in an efficient run-time environment;

3. the identification of areas where the compilation of FP programs can enhance the execution performance for pipelined computers, interconnections networks and non-von Neumann architectures.

6.2 Suggestions for Future Research

This section identifies areas that need to be further explored to complement and add to the results achieved in this work.

The implementation of the structural transformations for FP in this work was done using a string reduction semantics. The compiler described in Chapter 3 uses the string reduction model and also the performance results of Chapter 4 were based on a model that assumed string reduction. The same transformation techniques should also be implemented in a graph reduction model of execution for FP. Since graph reduction uses pointers to share partial results, different methods for the manipulation of the object descriptors must be used. The performance results of a graph reduction implementation are also expected to be different because of possible additional savings in memory requirements caused by the use of sharing.

One aspect that was neglected in this work was the problem of the syntax of FP. A great number of users will find that the FP syntax used here is cumbersome and does not expose the algorithms in a natural way. Some of these weaknesses, such as the syntax of expressions, can be considered minor, since there is no difficulty in changing the prefix notation used here to an infix notation that is considered to be more readable for humans. In fact, the FP interpreter presented in [Worl84] allows infix expressions.

Another syntactic aspect that deserves more work is the use of naming in functional languages. Since FP does not provide any facility for naming objects, they must be located by using selectors. This is the reason for the profusion of selectors in FP programs, as was determined in Section 2.2. When a programmer implements an algorithm in FP, he must first lay out the input object. Sub-objects are then referenced by position using selectors. The initial layout is an arbitrary choice, but after it is defined, the programmer must carefully follow it. If he wants to change the layout after the program was written he will have to change the selectors in the program. While some might argue that there is nothing wrong with this approach, it certainly is inconvenient, and some evidence suggests that it is as a major source of errors in the development of FP programs. Some work has been done in this area; for example, the interpreter cited above provides facility for the naming of input and output objects. Note that when naming is introduced in FP, care must be taken to preserve the semantics of the language. That is, by introducing naming one should not use it as variable names are used in procedural languages.

Another point for further work is the integration of the ideas presented in this work with ideas of other researchers that were described in Chapter 1. This integration could be attempted in the development of a real, production-style version of a compiler for FP or an FP-like language with the syntactic improvements discussed in the previous paragraph. The implementation of such a compiler should be attempted in conventional machines as well as is vector processing machines. The existence of these compilers will make it possible to compare their performance with today's preferred way of describing algorithms for computers, namely, procedural languages. We think that

full acceptance of the applicative style of programming by the software engineering community will occur only when such a production version of an FP system is developed.

The research described here could also be extended to other functional languages. Each proposed functional language has its own characteristics, and individual studies must be done to determine how applicable are the ideas presented here for each case. In particular, a good first candidate is the language Nial (Nested Interactive Array Language), presented in [Jenk86]. Although Nial was designed to support several styles of programming, for example, it has imperative constructs such as **for**, **while**, it encourages the applicative style of programming rather than imperative programming. Recursion and function applications using functional primitives, called operators in Nial, and functional forms, called transformers in Nial, are preferable to imperative control mechanisms. In fact, functional-style programs in Nial have a great similarity with FP programs. In Nial all data objects are treated as *nested arrays*. Atomic data is viewed as self-nesting arrays with no axes. Lists are arrays along one axis. It seems that the techniques developed in this work fit nicely in Nial and an integration of both approaches deserves further research.

REFERENCES

[Abra70] Abrams, P.S., *An APL Machine*, Ph.D. Dissertation, Stanford University, Stanford, CA (January 1970).

[Aho86] Aho, A.V., R. Sethi, and J.D. Ullman, *Compilers: Principles, Techniques and Tools*, Addison-Wesley (1986).

[Arno82] Arnold, C.N., "Performance Evaluation of Three Automatic Vectorizer Packages," *Proceedings of the 1982 International Conference on Parallel Processing*, pp.235-242 (August 24-27, 1982).

[Arvi82] Arvind and K.P. Gostelow, "The U-Interpreter," *IEEE Computer* **15**(2), pp.42-49 (February 1982).

[Arvi78] Arvind, K.P. Gostelow, and W. Plouffe, "An Asynchronous Programming Language and Computing Machine," Technical Report 114a, Department of Information and Computer Science, University of California, Irvine (December 1978).

[Arvi80] Arvind and R.E. Thomas, "I-Structures: An Efficient Data Type for Functional Languages," Technical Memo 178, MIT Laboratory for Computer Science , Cambridge, Mass. (September 1980).

[Augu84] Augustsson, L., "A Compiler for Lazy ML," *Conference Record of the 1984 Symposium on LISP and Functional Programming*, pp.218-227 (August 6-8, 1984).

[Back72] Backus, J., "Reduction Languages and Variable Free Programming," Research Report RJ 1010, IBM Yorktown Heights, NY (April 7, 1972).

[Back78] 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), pp.613-641 (August 1978).

[Bade83] Baden, S.B., *Berkeley FP User's Manual, Rev. 4.1*, Computer Science Division, University of California, Berkeley (March 25, 1983).

[Batc68] Batcher, K.E., "Sorting Networks and Their Applications," *Proceedings of the AFIPS SJCC* **32**, pp.307-314 (1968).

[Bell84] Bellegarde, F., "Rewriting Systems on FP Expressions that Reduce the Number of Sequences They Yield," *Conference Record of the 1984 Symposium on LISP and Functional Programming*, pp.63-73 (August 6-8, 1984).

[Berk75] Berkling, K.J., "Reduction Languages for Reduction Machines," *Proceedings of the Second Annual Meeting of Computer Architecture* , pp.133-138, IEEE (January 20-22, 1975).

[Budd84] Budd, T.A., "An APL Compiler for a Vector Processor," *ACM Transactions on Programming Languages and Systems* **6**(3), pp.297-313 (July 1984).

[Burr77] Burroughs-Corporation, "Burroughs Scientific Processor - Implementation of Fortran," Burroughs Document 6139E (1977).

[Burs77] Burstal, R.M. and J. Darlington, "A Transformation System for Developing Recursive Programs," *Journal of the ACM* **24**(1), pp.44-67 (January 1977).

[Burs80] Burstall, R.M., D.B. MacQueen, and D.T. Sannella, "HOPE: An Experimental Applicative Language," *LISP Conference Record*, pp.136-143 (1980).

[Cart85] Cartwright, R., "Types as Intervals," *Conference Record of the Twelfth Annual Symposium on Principles of Programming Languages*, pp.22-36 (January 14-16, 1985).

[Clar80] Clarke, T.J.W., P.J.S. Gladstone, C.D. MacLean, and A.C. Norman, "SKIM - The S, K, I Reduction Machine," *Proceedings of the LISP-80 Conference*, pp.128-135 (1980).

[Cray76] Cray-Corporation, "CRAY-1 Computer System: Reference Manual," Cray Research Publication 2240004 (1976).

[Darl81] Darlington, J. and M. Reeve, "ALICE - A Multiprocessor Reduction Machine for the Parallel Evaluation of Applicative Languages," *ACM Conference on Functional Languages and Architectures*, pp.65-75 (1981).

[Denn74] Dennis, J.D., "First Version of a Data Flow Procedure Language," *Lecture Notes in Computer Science* **19**, pp.362-376, Springer-Verlag (1974).

[Denn80] Dennis, J.D., "Data Flow Supercomputers," *IEEE Computer* **13**(11), pp.48-56 (November 1980).

[Gaud82] Gaudiot, J.-L., *On Program Decomposition and Partitioning in Data Flow Systems*, Ph.D. Dissertation, Computer Science Department, University of California, Los Angeles, CA (December 1982).

[Gaud85] Gaudiot, J.-L. and M.D. Ercegovac, "Performance Evaluation of a Simulated Data-Flow Computer with Low-Resolution Actors," *Journal of Parallel and Distributed Computing* **2**, pp.321-351 (1985).

[Givl84] Givler, J.S. and R.B. Kieburtz, "Schema Recognition for Program Transformations," *Conference Record of the 1984 Symposium on LISP and Functional Programming*, pp.74-84 (August 6-8, 1984).

[Guib78] Guibas, L.J. and D.K. Wyatt, "Compilation and Delayed Evaluation in APL," *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, pp.1-8 (January 23-25, 1978).

[Gurd85] Gurd, J.R., C.C. Kirkham, and I. Watson, "The Manchester Prototype Dataflow Computer," *Communications of the ACM* **28**(1) (January 1985).

[Hock81] Hockney, R.W. and C.R. Jesshope, *Parallel Computers*, Adam Hilger Ltd, Bristol (1981).

[Huda85] Hudak, P. and A. Bloss, "The Aggregate Update Problem in Functional Programming Systems," *Conference Record of the 12th Annual ACM Symposium on Principles of Programming Languages*, pp.300-314 (January 14-16, 1985).

[Isla81] Islam, N., T.J. Myers, and P. Broome, "A Simple Optimizer for FP-like Languages," *Proceedings of the 1981 ACM Conference on Functional Programming Languages and Computer Architecture*, pp.33-39 (October 18-22, 1981).

[Iver62] Iverson, K.E., *A Programming Language*, John Wiley and Sons, New York, NY (1962).

[Jenk86] Jenkins, M.A., J.I. Glasgow, and C.D. McCrosky, "Programming Styles in Nial," *IEEE Software* 3(1), pp.46-55 (January 1986).

[John75] Johnson, S.C., "Yacc: Yet Another Compiler Compiler," Computing Science Technical Report No. 32, Bell Laboratories, Murray Hill, NJ (1975).

[Kasc79] Kascic, M.J. Jr., "Vector Processing on the CYBER 200," pp. 237-270 in *Infotech State of the Art Report: Supercomputers - Vol. 2*, ed. C.R. Jesshope and R.W. Hockney, Infotech Intl. Ltd. (1979).

[Kata84] Katayama, T., "Type Inference and Type Checking for Functional Programming Languages - A Reduced Computation Approach," *Conference Record of the 1984 Symposium on LISP and Functional Programming*, pp.263-272 (August 6-8, 1984).

[Kell79] Keller, R.M., G. Lindstrom, and S.S. Patil, "A Loosely-Coupled Applicative Multiprocessing System," *Proceedings of the AFIPS NCC 1979*, pp.613-622, AFIPS Press, Montvale, NJ (June 4-7, 1979).

[Kell83] Kellman, J.N., *Parallel Execution of Functional Programs*, M.Sc. Dissertation, Computer Science Department, University of California, Los Angeles, CA (January 1983).

[Kern78] Kernighan, B.W. and D.M. Ritchie, *The C Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1978).

[Kieb81] Kieburtz, R.B. and J. Shultis, "Transformation of FP Program Schemes," *Proceedings of the 1981 ACM Conference on Functional Programming Languages and Computer Architecture*, pp.41-48 (October 18-22, 1981).

[Kuck78] Kuck, D., *The Structure of Computers and Computations - Volume 1*, John Wiley & Sons, Inc. (1978).

[Kuck81] Kuck, D.J., R.H. Kuhn, D.A. Padua, B. Leasure, and M. Wolfe, "Dependence Graphs and Compiler Optimization," *Proceedings of the 8th ACM Symposium on the Principles of Programming Languages*, pp.207-218 (January 1981).

[Lesk75] Lesk, M.E. and E. Schmidt, "Lex - A Lexical Analyzer Generator," Computing Science Technical Report No. 39, Bell Laboratories, Murray Hill, NJ (October 1975).

[Mago84a] Mago, G. and D. Middleton, "The FFP Machine - A Progress Report," *Proceedings of the International Workshop on High-Level Computer Architecture 84*, pp.5.13-5.25 (May 21-25, 1984).

[Mago79] Mago, G.A., "A Network of Microprocessors to Execute Reduction Languages," *International Journal of Computer and Information Science* **8**(5, 6), pp.349-385, 435-471 (1979). (in two parts).

[Mago80] Mago, G.A., "A Cellular Computer Architecture for Functional Programming," *Proceedings of the COMPCON Fall 1980*, pp.179-187 (1980).

[Mago81] Mago, G.A., "Copying Operands Versus Copying Results: A Solution to the Problem of Large Operands in FFP's," *Proceedings of the 1981 ACM Conference on Functional Programming Languages and Computer Architecture*, pp.93-97 (October 18-22, 1981).

[Mago84] Mago, G.A., D.F. Stanat, and A. Koster, "Program Execution on a Cellular Computer: Some Matrix Algorithms," Unpublished Draft (1984).

[Mann75] Manna, Z. and R. Waldinger, "Knowledge and Reasoning in Program Sysnthesis," *Artificial Intelligence* **6**, p.175 (1975).

[Mann79] Manna, Z. and R. Waldinger, "Synthesis: Dreams => Programs," *IEEE Transactions on Software Engineering* **SE-5**(4), pp.157-164 (July 1979).

[McCa60] McCarthy, J., "Recursive Functions of Symbolic Expressions and their Computation by Machine," *Communications of the ACM* 3(4), pp.184-195 (April 1960).

[McGr79] McGraw, J.R., "Data Flow Computing: Software Development," *Proceedings of the IEEE International Conference on Distributed Systems*, pp.242-251 (1979).

[McGr83] McGraw, J.R., "SISAL - Streams and Iteration in a Single-Assignment Language," Language Reference Manual (version 1.0), Lawrence Livermore Laboratory , Livermore, CA (July 1983).

[Mint76] Minter, C.R., "A Machine Design for Efficient Implementation of APL," Research Report 81, Yale University, New Haven, CT (1976).

[Mish85] Mishra, P. and U.S. Reddy, "Declaration-Free Type Checking," *Conference Record of the Twelfth Annual Symposium on Principles of Programming Languages*, pp.7-21 (January 14-16, 1985).

[Morr80] Morris, J.H., E. Schmidt, and P.L. Wadler, "Experience with an Applicative String Processing Language," *Proceedings of the ACM Symposium on Principles of Programming Languages*, pp.32-46 (July 1980).

[Padu80] Padua, D.A., D.J. Kuck, and D.H. Lawrie, "High-Speed Multiprocessors and Compilation Techniques," *IEEE Transactions on Computers* C-29(9), pp.763-776 (September 1980).

[Park80] Parker, D.S., "Notes on Shuffle/Exchange-Type Switching Networks," *IEEE Transactions on Computers* C-29(3), pp.213-222 (March 1980).

[Paul75] Paul, G. and M.W. Wilson, "The Vectran Language: An Experimental Language for Vector/Matrix Array Processing," IBM Research Report 320-34 (1975).

[Pend86] Pendergrast, J.S. and B.G. Ryder, "FPOPT: A Globally Optimizing Compiler for FP," Technical Report DCS-TR-175, Rutgers University, Department of Computer Science, New Brunswick, NJ (March 1986).

[Perr79] Perrott, R.H., "A Language for Array and Vector Processors," *ACM Transactions on Programming Languages and Systems* 1(2), pp.177-195 (October 1979).

[Pett84] Pettorossi, A., "A Powerful Strategy for Deriving Efficient Programs by Transformation," *Conference Record of the 1984 Symposium on LISP and Functional Programming*, pp.273-281 (August 6-8, 1984).

[Plai85] Plaisted, D.A., "An Architecture for Fast Data Movement in the FFP Machine," *Conference on Functional Programming Languages and Computer Architecture*, pp.147-163, Lecture Notes in Computer Science N. 201,

Springer-Verlag (September 16-19, 1985).

[Ravi86a] Ravi, T.M., *Partitioning and Allocation of Functional Programs for Data Flow Processors*, M.Sc. Thesis, Computer Science Department, University of California, Los Angeles (April 1986).

[Ravi86] Ravi, T.M. and M.D. Ercegovac, "Allocation for the SANDAC Multiprocessor System," Technical Report No. CSD-860059, Computer Science Department, University of California, Los Angeles, CA (February 1986).

[Riga84] Riganati, J.P. and P.B. Schneck, "Supercomputing," *IEEE Computer* **17**(10), pp.97-113 (October 1984).

[Rumb77] Rumbaugh, J.E., "A Data Flow Multiprocessor," *IEEE Transactions on Computers* **C-26**(2), pp.138-146 (February 1977).

[Sain84] Saint-James, E., "Recursion is More Efficient than Iteration," *Conference Record of the 1984 Symposium on LISP and Functional Programming*, pp.228-234 (August 6-8, 1984).

[Schl84] Schlag, M.D.F., "Extracting Geometry from FP for VLSI Layout," Technical Report No. CSD-840043, Computer Science Department, University of California, Los Angeles (October 1984).

[Stan81] Stanat, D.F. and G.A. Mago, "Optimal Storage Management in a Cellular Computer," Technical Report No. 81-006, Department of Computer Science, University of North Carolina at Chapel Hill (1981).

[Stev75] Stevens, K.G. Jr., "CFD - A Fortran-like Language for the ILLIAC-IV," *SIGPLAN Notices* **10**(3), pp.72-80 (1975).

[Syre77] Syre, J.C., D. Comte, and N. Hifdi, "Pipelining, Parallelism and Asynchronism in the LAU System," *Proceedings of the 1977 International Conference on Parallel Processing*, pp.87-92 (August 1977).

[Thom78] Thompson, C.D., "Generalized Connection Networks for Parallel Processor Intercommunication," *IEEE Transactions on Computers* **C-27**(12), pp.1119-1125 (December 1978).

[Trel82] Treleaven, P.C., D.R. Brownbridge, and R.P. Hopkins, "Data-Driven and Demand-Driven Computer Architecture," *ACM Computing Surveys* **14**(1), pp.93-143 (March 1982).

[Turn79] Turner, D.A., "A New Implementation Technique for Applicative Languages," *Software - Practice and Experience* 9(1), pp.31-49 (January 1979).

[Turn81] Turner, D.A., "The Semantic Elegance of Applicative Languages," *Proceedings of the 1981 ACM Conference on Functional Programming Languages and Architectures*, pp.85-92 (October 18-22, 1981).

[Turn82] Turner, D.A., "Recursion Equations as a Programming Language," pp. 1-28 in *Functional Programming and Its Applications: An Advanced Course*, Cambridge University, Cambridge, England (1982).

[Wadl81] Wadler, P., "Applicative Style Programming, Program Transformation, and List Operators," *Proceedings of the 1981 ACM Conference on Functional Programming Languages and Computer Architecture*, pp.25-32 (October 18-22, 1981).

[Wadl84] Wadler, P., "Listlessness is Better than Laziness," *Conference Record of the 1984 Symposium on LISP and Functional Programming*, pp.45-52 (August 6-8, 1984).

[Waks68] Waksman, A., "A Permutation Network," *Journal of the ACM* 9(1), pp.159-163 (January 1968).

[Wass82] Wasserman, A.I. and S. Gutz, "The Future of Programming," *Communications of the ACM* **25**(3), pp.196-206 (March 1982).

[Will81] Williams, E.H. Jr., *Analysis of FFP Programs for Parallel Associative Searching*, Ph.D. Dissertation, University of North Carolina at Chapel Hill (1981).

[Will82] Williams, J.H., "On the Development of the Algebra of Functional Programs," *ACM Transactions on Programming Languages and Systems* 4(4), pp.733-757 (October 1982).

[Wolf81] Wolfe, M. and B. Leasure, *Understanding KAP Output Listings*, Kuck and Associates, Inc. (September 1981).

[Worl84] Worley, J., "The UCLA T-FP User Manual," , Internal Memorandum, UCLA Computer Science Department, Los Angeles, CA (June 1984).

APPENDIX 1
DESCRIPTION OF FP

Objects

The set of objects Ω consists of the atoms and sequences $<x_1, x_2, \ldots, x_k>$ (where $x_i \in \Omega$). (Lisp users should note the similarity to the list structure syntax; just replace the parenthesis by angle brackets and commas by blanks. There are no 'quoted' objects, i.e., 'abc). The atoms uniquely determine the set of valid objects and consist of the numbers, quoted ascii strings ("abcd"), and unquoted alphanumeric strings (abc3). There are three predefined atoms, **T** and **F**, that correspond to the logical values 'true' and 'false', and the undefined atom **?**, *bottom*. *Bottom* denotes the value returned as the result of an undefined operation, e.g., division by zero. The empty sequence, <>, is also an atom. The following are examples of valid FP objects:

?	1.47	3888888888888
ab	*"CD"*	<1,<2,3>>
<>	T	<*a*, <>>

There is one restriction on object construction: no object may contain the undefined atom, such an object is itself undefined, e.g., <1, ?> ≡ ?. This

property is the so-called "bottom preserving property".

Application

This is the single FP operation and is designated by the colon (":"). For a function σ and an object x, $\sigma:x$ is an application and its meaning is the object that results from applying σ to x (i.e., evaluating $\sigma(x)$). We say that σ is the *operator* and that x is the *operand.* The following are examples of applications:

$$+:\langle 7,8\rangle \equiv 15 \qquad \mathbf{tl}:\langle 1,2,3\rangle \equiv \langle 2,3\rangle$$
$$\mathbf{1}:\langle a,b,c,d\rangle \equiv a \qquad \mathbf{2}:\langle a,b,c,d\rangle \equiv b$$

Functions

All functions (F) map objects into objects, moreover, they are *strict*:

$$\sigma:? \equiv ?, \quad \forall \sigma \in F$$

To formally characterize the primitive functions, we use a modification of McCarthy's conditional expressions [McCa60]:

$$p_1 \rightarrow e_1 \; ; \; \cdots \; ; p_n \rightarrow e_n \; ; \; e_{n+1}$$

This statement is interpreted as follows: return function e_1 if the predicate p_1 is true , . . . , e_n if p_n is true. If none of the predicates are satisfied then default to e_{n+1}. It is assumed that x, x_i, y, y_i, $z_i \in \Omega$.

Selector Functions

For a nonzero integer μ,

$\mu : x \equiv$

$x = \langle x_1, x_2, \ldots, x_k \rangle \wedge 0 < \mu \leq k \rightarrow x_\mu$;

$x = \langle x_1, x_2, \ldots, x_k \rangle \wedge -k \leq \mu < 0 \rightarrow x_{k+\mu+1}$; ?

pick : $\langle n,x \rangle \equiv$

$x = \langle x_1, x_2, \ldots, x_k \rangle \wedge 0 < n \leq k \rightarrow x_n$;

$x = \langle x_1, x_2, \ldots, x_k \rangle \wedge -k \leq n < 0 \rightarrow x_{k+n+1}$; ?

The user should note that the function symbols **1,2,3,...** are to be distinguished from the atoms 1,2,3,....

last : $x \equiv$

$x = \langle\rangle \rightarrow \langle\rangle$;

$x = \langle x_1, x_2, \ldots, x_k \rangle \wedge k \geq 1 \rightarrow x_k$; ?

first : $x \equiv$

$x = \langle\rangle \rightarrow \langle\rangle$;

$x = \langle x_1, x_2, \ldots, x_k \rangle \wedge k \geq 1 \rightarrow x_1$; ?

Tail Functions

tl : $x \equiv$

$x = \langle x_1 \rangle \rightarrow \langle\rangle$;

$x = \langle x_1, x_2, \ldots, x_k \rangle \wedge k \geq 2 \rightarrow \langle x_2, \ldots, x_k \rangle$; ?

tlr : $x \equiv$

$x = \langle x_1 \rangle \rightarrow \langle\rangle$;

$x = \langle x_1, x_2, \ldots, x_k \rangle \wedge k \geq 2 \rightarrow \langle x_1, \ldots, x_{k-1} \rangle$; ?

Note: There is also a function **front** that is equivalent to **tlr**.

Distribute from left and right

distl : $x \equiv$

$x = \langle y, \langle\rangle \rangle \rightarrow \langle\rangle$;

$x = \langle y, \langle z_1, z_2, \ldots, z_k \rangle \rangle \rightarrow \langle \langle y, z_1 \rangle, \ldots, \langle y, z_k \rangle \rangle$; ?

distr : $x \equiv$

$x = \langle \langle\rangle, y \rangle \rightarrow \langle\rangle$;

$x = \langle \langle y_1, y_2, \ldots, y_k \rangle, z \rangle \rightarrow \langle \langle y_1, z \rangle, \ldots, \langle y_k, z \rangle \rangle$; ?

Identity

id : $x \equiv x$

out : $x \equiv x$

Out is similar to **id**. Like **id** it returns its argument as the result; unlike **id** it prints its result on *stdout*. It is the only function with a side effect. **Out** is intended to be used for debugging only.

Append left and right

apndl : $x \equiv$

$x = \langle y, \langle\rangle \rangle \rightarrow \langle y \rangle$;

$x = \langle y, \langle z_1, z_2, \ldots, z_k \rangle \rangle \rightarrow \langle y, z_1, z_2, \ldots, z_k \rangle$; ?

apndr : $x \equiv$

$x = \langle\langle\rangle, z\rangle \rightarrow \langle z\rangle$;

$x = \langle\langle y_1, y_2, \ldots, y_k\rangle, z\rangle \rightarrow \langle y_1, y_2, \ldots, y_k, z\rangle$; ?

Transpose

trans : $x \equiv$

$x = \langle\langle\rangle, \ldots, \langle\rangle\rangle \rightarrow \langle\rangle$;

$x = \langle x_1, x_2, \ldots, x_k\rangle \rightarrow \langle y_1, \ldots, y_m\rangle$; ?

where $x_i = \langle x_{i1}, \ldots, x_{im}\rangle \wedge y_j = \langle x_{1j}, \ldots, x_{kj}\rangle$,

$1 \leq i \leq k$, $1 \leq j \leq m$.

Reverse

reverse : $x \equiv$

$x = \langle\rangle \rightarrow ; \langle\rangle$

$x = \langle x_1, x_2, \ldots, x_k\rangle \rightarrow \langle x_k, \ldots, x_1\rangle$; ?

Rotate left and right

rotl : $x \equiv$

$x = \langle\rangle \rightarrow \langle\rangle$; $x = \langle x_1\rangle \rightarrow \langle x_1\rangle$;

$x = \langle x_1, x_2, \ldots, x_k\rangle \wedge k \geq 2 \rightarrow \langle x_2, \ldots, x_k, x_1\rangle$; ?

rotr : $x \equiv$

$x = \langle\rangle \rightarrow \langle\rangle$; $x = \langle x_1\rangle \rightarrow \langle x_1\rangle$;

$x = \langle x_1, x_2, \ldots, x_k\rangle \wedge k \geq 2 \rightarrow \langle x_k, x_1, \ldots, x_{k-2}, x_{k-1}\rangle$; ?

Concatenate

concat : $x \equiv$

$x = <<x_{11}, \ldots, x_{1k}>, <x_{21}, \ldots, x_{2n}>, \ldots, <x_{m1}, \ldots, x_{mp}>>$

$\wedge\ k, m, n, p > 0 \rightarrow$

$<x_{11}, \ldots, x_{1k}, x_{21}, \ldots, x_{2n}, \ldots, x_{m1}, \ldots, x_{mp}>$; ?

Concatenate removes all occurrences of the null sequence, e.g.,

concat : <<1,3>,<>,<2,4>,<>,<5>> ≡ <1,3,2,4,5>

Pair and Split

pair : $x \equiv$

$x = <x_1, x_2, \ldots, x_k> \wedge\ k>0 \wedge\ k$ is even $\rightarrow$

$<<x_1,x_2>, \ldots, <x_{k-1},x_k>>$;

$x = <x_1, x_2, \ldots, x_k> \wedge\ k>0 \wedge\ k$ is odd $\rightarrow$

$<<x_1,x_2>, \ldots, <x_k>>$; ?

split : $x \equiv$

$x = <x_1> \rightarrow <<x_1>,<>>$;

$x = <x_1, x_2, \ldots, x_k> \wedge\ k>1 \rightarrow$

$<<x_1, \ldots, x_{\lceil k/2 \rceil}>, <x_{\lceil k/2 \rceil+1}, \ldots, x_k>>$; ?

List of integers

iota : $x \equiv$

$x = 0 \rightarrow <>$;

$x \in N^+ \rightarrow <1,2,\ldots,x>$; ?

Predicate (Test) Functions

atom : $x \equiv x \in atoms \rightarrow \mathbf{T}; x \neq ? \rightarrow \mathbf{F}; ?$

eq : $x \equiv x = \langle y,z \rangle \wedge y = z \rightarrow \mathbf{T}; x = \langle y,z \rangle \wedge y \neq z \rightarrow \mathbf{F}; ?$

Also less than (<), greater than (>), greater than or equal (>=), less than or equal (<=), not equal (~=); = is a synonym for **eq**.

null : $x \equiv x = \langle\rangle \rightarrow \mathbf{T}; x \neq ? \rightarrow \mathbf{F}; ?$

length : $x \equiv x = \langle x_1, x_2, \ldots, x_k \rangle \rightarrow k; x = \langle\rangle \rightarrow 0; ?$

Arithmetic/Logical

+ : $x \equiv x = \langle y,z \rangle \wedge y,z$ are numbers $\rightarrow y+z; ?$

− : $x \equiv x = \langle y,z \rangle \wedge y,z$ are numbers $\rightarrow y-z; ?$

* : $x \equiv x = \langle y,z \rangle \wedge y,z$ are numbers $\rightarrow y \times z; ?$

/ : $x \equiv x = \langle y,z \rangle \wedge y,z$ are numbers $\wedge z \neq 0 \rightarrow y \div z; ?$

and : $\langle x,y \rangle \equiv x = \mathbf{T} \rightarrow y; x = \mathbf{F} \rightarrow \mathbf{F}; ?$

or : $\langle x,y \rangle \equiv x = \mathbf{F} \rightarrow y; x = \mathbf{T} \rightarrow \mathbf{T}; ?$

xor : $\langle x,y \rangle \equiv$

$x=\mathbf{T} \wedge y=\mathbf{T} \rightarrow \mathbf{F}; x=\mathbf{F} \wedge y=\mathbf{F} \rightarrow \mathbf{F};$

$x=\mathbf{T} \wedge y=\mathbf{F} \rightarrow \mathbf{T}; x=\mathbf{F} \wedge y=\mathbf{T} \rightarrow \mathbf{T}; ?$

not : $x \equiv x = \mathbf{T} \rightarrow \mathbf{F}; x = \mathbf{F} \rightarrow \mathbf{T}; ?$

Library Routines

sin : $x \equiv x$ is a number $\rightarrow \sin(x)$; ?

asin : $x \equiv x$ is a number $\wedge\ |x| \leq 1 \rightarrow \sin^{-1}(x)$; ?

cos : $x \equiv x$ is a number $\rightarrow \cos(x)$; ?

acos : $x \equiv x$ is a number $\wedge\ |x| \leq 1 \rightarrow \cos^{-1}(x)$; ?

exp : $x \equiv x$ is a number $\rightarrow e^x$; ?

log : $x \equiv x$ is a positive number $\rightarrow \ln(x)$; ?

mod : $\langle x,y \rangle \equiv x$ and y are numbers $\rightarrow x - y \times \lfloor x/y \rfloor$; ?

Functional Forms

Functional forms define new *functions* by operating on function and object *parameters* of the form. The resultant expressions can be compared and contrasted to the *value*-oriented expressions of traditional programming languages. The distinction lies in the domain of the operators; functional forms manipulate functions, while traditional operators manipulate values.

One functional form is *composition*. For two functions ϕ and ψ the form ϕ @ ψ denotes their composition:

$$(\phi \text{ @ } \psi) : x \equiv \phi:(\psi:x),\ \forall\ x \in \Omega$$

The *constant* function takes an object parameter:

$$\%x:y \equiv y = ? \rightarrow ?;\ x, \quad \forall\ x,y \in \Omega$$

The function %? always returns ?.

In the following description of the functional forms, we assume that ξ, ξ_i, σ, σ_i, τ, and τ_i are functions and that x, x_i, y are objects.

Composition

$$(\sigma\ @\ \tau):x \equiv \sigma:(\tau:x)$$

Construction

$$[\sigma_1, \ldots, \sigma_n]:x \equiv <\sigma_1:x, \ldots, \sigma_n:x>$$

Note that construction is also bottom-preserving, e.g.,

$$[+,/]:<3,0> = <3,?> = ?$$

Conditional

$$(\xi \text{ -> } \sigma; \tau):x \equiv$$
$$(\xi:x) = \mathbf{T} \rightarrow \sigma:x;$$
$$(\xi:x) = \mathbf{F} \rightarrow \tau:x; ?$$

The reader should be aware of the distinction between *functional expressions*, in the variant of McCarthy's conditional expression, and the *functional form* introduced here. In the former case the result is a *value*, while in the latter case the result is a *function*. Unlike Backus' FP, the conditional

form *must* be enclosed in parenthesis, e.g.,

(isNegative -> - @ [%0,id] ; id)

Constant

$\%x : y \equiv y = ? \rightarrow ?; x, \qquad \forall x \in \Omega$

This function returns its object parameter as its result.

Right Insert

$!\sigma : x \equiv$

$x = <> \rightarrow e_f : x;$

$x = <x_1> \rightarrow x_1;$

$x = <x_1, x_2, \ldots, x_k> \wedge k > 2 \rightarrow \sigma:<x_1, !\sigma:<x_2, \ldots, x_k>>; ?$

e.g., $!+:<4,5,6> = 15.$

If σ has a right identity element e_f, then $!\sigma:<> = e_f$, e.g.,

$!+ : <> = 0$ and $!* : <> = 1$

Currently, identity functions are defined for + (0), - (0), * (1), / (1); also for **and** (T), **or** (F), **xor** (F). All other unit functions default to bottom (?).

Tree Insert

$|\sigma : x \equiv$

$x = <> \rightarrow e_f : x;$

$x = <x_1> \rightarrow x_1;$

$x = <x_1, x_2, \ldots, x_k> \wedge k > 1 \rightarrow$

$\sigma : <| \sigma : <x_1, \ldots, x_{\lceil k/2 \rceil}> , | \sigma : <x_{\lceil k/2 \rceil+1}, \ldots, x_k>>; ?$

e.g.,

$| +:<4,5,6,7> \equiv +:<+:<4,5>,+:<6,7>> \equiv 15$

Tree insert uses the same identity functions as right insert.

Apply to All

$\& \sigma: x \equiv$

$x = <> \rightarrow <>;$

$x = <x_1, x_2, \ldots, x_k> \rightarrow <\sigma:x_1, \ldots, \sigma:x_k>; ?$

User Defined Functions

An FP definition is entered as follows:

{*fn-name* *fn-form*},

where *fn-name* is an ascii string consisting of letters, numbers and the underline symbol, and *fn-form* is any valid functional form, including a single primitive or defined function. For example the function

{*factorial* !* @ iota}

is the non-recursive definition of the factorial function. Since FP systems are applicative it is permissible to substitute the actual definition of a function for any reference to it in a functional form: if $f \equiv 1@2$ then $f : x \equiv 1@2 : x, \quad \forall x \in \Omega.$

References to undefined functions bottom out:

$$f : x \equiv ? \quad \forall x \in \Omega, f \notin F$$

APPENDIX 2

SYMBOLIC STRUCTURAL TRANSFORMATIONS FOR FP

In this appendix we describe the basic *structural transformations* induced by each FP primitive. A summary of these transformations was presented in Chapter 2. The transformations induced by the FP *functional forms* are completely described in Chapter 2. In the description, the notation $f : s \rightarrow t$ means that a FP function f applied to an object of structure s returns an object of structure t.

Selectors: **k**: $<s_1, s_2, \cdots, s_n>$ and $1 \leq k \leq n \rightarrow s_k$; Λ

For homogeneous sequences:

k: $<s^{1:n}>$ and $1 \leq k \leq n \rightarrow s^{k:k}$; Λ

last: $<> \rightarrow <>$;

$<s_1, s_2, \cdots, s_n>$ and $n \geq 1 \rightarrow s_n$; Λ

For homogeneous sequences:

last: $<s^{1:n}>$ and $n \geq 1 \rightarrow s^{n:n}$; Λ

first: $<> \rightarrow <>$;

$<s_1, s_2, \cdots, s_n>$ and $n \geq 1 \rightarrow s_1$; Λ

For homogeneous sequences:

first: $<s^{1:n}>$ and $n \geq 1 \rightarrow s^{1:1}$; Λ

tl: $\langle s \rangle \rightarrow \langle\rangle$;

$\langle s_1, s_2, \cdots, s_n \rangle$ and $n \geq 2 \rightarrow \langle s_2, s_3, \cdots, s_n \rangle$; Λ

For homogeneous sequences:

tl: $\langle s^{1:n} \rangle$ and $n \geq 2 \rightarrow \langle s^{2:n} \rangle$; Λ

tlr: $\langle s \rangle \rightarrow \langle\rangle$;

$\langle s_1, s_2, \cdots, s_n \rangle$ and $n \geq 2 \rightarrow \langle s_1, s_2, \cdots, s_{n-1} \rangle$; Λ

For homogeneous sequences:

tlr: $\langle s^{1:n} \rangle$ and $n \geq 2 \rightarrow \langle s^{1:n-1} \rangle$; Λ

distl: $\langle s, \langle\rangle \rangle \rightarrow \langle\rangle$;

$\langle s, \langle t^{1:n} \rangle \rangle \rightarrow \langle \langle s, t \rangle^{1:n} \rangle$; Λ

Restriction: Second element is a homogeneous sequence.

distr: $\langle \langle\rangle , t \rangle \rightarrow \langle\rangle$;

$\langle \langle s^{1:n} \rangle, t \rangle \rightarrow \langle \langle s\ t \rangle^{1:n} \rangle$; Λ

Restriction: First element is a homogeneous sequence.

id: $s \rightarrow s$.

apndl: $\langle s, \langle\rangle \rangle \rightarrow \langle s \rangle$;

$\langle s, \langle t_1, t_2, \cdots, t_n \rangle \rangle \rightarrow \langle s, t_1, t_2, \cdots, t_n \rangle$; Λ

For homogeneous sequences:

$\langle s, \langle t^{1:n} \rangle \rangle \rightarrow \langle s, t^{1:n} \rangle$; Λ

Special case:

$\langle t, \langle t^{1:n} \rangle \rangle \rightarrow \langle t^{1:n+1} \rangle$; Λ

apndr: $\langle \langle\rangle, s \rangle \rightarrow \langle s \rangle$;

$< <s_1, s_2, \cdots, s_n>, t > \rightarrow < s_1, s_2, \cdots, s_n, t >; \Lambda$

For homogeneous sequences:

$< <s^{1:n}>, t > \rightarrow < s^{1:n}, t >; \Lambda$

Special case:

$< <s^{1:n}>, s > \rightarrowtail < s^{1:n+1} >; \Lambda$

trans: $< <>^n> \rightarrow <>, \quad n \geq 1;$

$< <s^{1:m}>^{1:n} > \rightarrow < <s^{1:n}>^{1:m} > \quad m, n \geq 1; \quad \Lambda$

Restriction: Homogeneous sequences.

reverse: $<> \rightarrow <>;$

$< s^n > \rightarrow < s^n >; \Lambda$

Restriction: Homogeneous sequence under Representation 1.

rotl: $<> \rightarrow <>;$

$< s^n > \rightarrow < s^n >; \Lambda$

Restriction: Homogeneous sequence under Representation 1.

rotr: $<> \rightarrow <>;$

$< s^n > \rightarrow < s^n >; \Lambda$

Restriction: Homogeneous sequence under Representation 1.

concat:

$< <s_{11}, \cdots, s_{1k}>, \cdots, <s_{m1}, \cdots, s_{mn}> >, \quad k, m, n > 0 \rightarrow$

$< s_{11}, \cdots, s_{1k}, \cdots, s_{m1}, \cdots, s_{mn}>; \Lambda$

concat removes all occurrences of the null sequence.

pair: $< s^n > \rightarrow < <s^2>^{n/2} >; \Lambda$

Restrictions: Homogeneous sequence and n even.

split: $< s^{1:n} > \rightarrow < < s^{1:n/2} > < s^{n/2+1:n} > >;\ \Lambda$

Restrictions: Homogeneous sequence and n even.

atom: $s \rightarrow a.$

eq: $< s,\ t > \rightarrow a.$

Moreover, if $s \neq t$, **eq:** $< s,\ t > \rightarrow$ **F**.

<, >, >=, <=, !=: $< a^2 > \rightarrow a;\ \Lambda$

null: $<> \rightarrow$ **T; F**

length: $< s_1, s_2, \cdots, s_n > \rightarrow n;$

$<> \rightarrow 0;\ \Lambda$

For homogeneous sequences:

length: $< s^{1:n} > \rightarrow n;\ \Lambda$

+, -, *, / : $< a^2 > \rightarrow a;\ \Lambda$

and, or, xor: $< a^2 > \rightarrow a;\ \Lambda$

not: $a \rightarrow a;\ \Lambda$

iota: $a \rightarrow < a^n >$, where $n > 0$, n integer; Λ

In general, n is indeterminate. However, **iota** can be solved if applied to a (compile time) constant:

iota @ %3: x ≡ <1 2 3>

iota @ length: <a b c> ≡ <1 2 3>

pick: $<a, <s^n> > \rightarrow s$

Restriction: Homogeneous sequences under Representation 1.

(Value of a is indeterminate).

APPENDIX 3

FP IMPLEMENTATION OF THE LAWRENCE LIVERMORE LABORATORY KERNELS

In the next pages we show the FP implementation of 10 of the 14 Livermore Kernels as described in [Riga84]. For each kernel, the original FORTRAN description is presented first, followed by the FP description. The FP implementations expose the maximum parallelism present in each loop.

The kernels 4, 8, 13 and 14 are not shown because they contain indirect addressing in the form of A (B(I)). The FP implementation of this type of FORTRAN addressing, as was discussed in Chapter 5, implies use of the FP primitive **pick** which is not solved by the compilation techniques described in Chapter 3.

Kernel 1: 1-dimensional hydrodynamics excerpt

FORTRAN:

```
DO 1 K = 1, 400
  X(K) = Q + Y(K) * (R*Z(K+10) + S*Z(K+11))
1 CONTINUE
```

```
FP: input object: < Y(1:k), Z(1:k+11) >
    output object: X(1:k)

{Zkplus10  tl @ tl @ tl @ tl @ tl @ tl @ tl @ tl @ tl @ tl @ tlr @ 2 }  # select Z(11:k+10)

{Zkplus11  tl @ tl @ tl @ tl @ tl @ tl @ tl @ tl @ tl @ tl @ tl @ 2 }  # select Z(12:k+11)

{Yk  1}  # select Y(1:k)

{R %1}   # constants
{S %1}   # (arbitrary values are assigned here)
{Q %1}

{align   [Yk, Zkplus10, Zkplus11] }   # align operands

{Xk   + @ [Q, * @ [1, + @ [* @ [R, 2], * @ [S, 3] ] ] ] }   # expression (Xk)

# LOOP:

{LLL1   &Xk @      # all at once
        trans @    # put operands together
        align
}
```

Figure A3.1 - Livermore Kernel No. 1

Kernel 2: Unrolled Inner Product

```
DO 2 K = 1, 996, 5
 Q = Q + Z(K)*X(K) + Z(K+1)*X(K+1) + Z(K+2)*X(K+2)
     + Z(K+3)*X(K+3) + Z(K+4)*X(K+4)
2 CONTINUE
```

Kernel 3: Inner Product

```
DO 3 K = 1, 1000
 Q = Q + Z(K)*X(K)
3 CONTINUE
```

FP: Both kernels have the same FP implementation
input object: < Z(1:n), X(1:n) >
output object: Q

{LLL2 (!+) @ &* @ trans}

Figure A3.2 - Livermore Kernels Nos. 2 and 3

Kernel 5: Tri-diagonal elimination, below diagonal

FORTRAN:

```
DO 5 J = 2, 998, 3
  X(I) = Z(I) * (Y(I)-X(I-1))
  X(I-1) = Z(I+1) * (Y(I+1) - X(I))
  X(I-2) = Z(I+2) * (Y(I+2) - X(I+1))
5 CONTINUE
```

this loop is equivalent to:

```
DO 5 I = 2, 1000
  X(I) = Z(I) * (Y(I) - X(I-1))
5 CONTINUE
```

FP: Input object: < <x1> Y(1:n) Z(1:n) >
Output object: X(2:n)

```
{setup [1, tl@2, tl@3]}

{recur (null@2 -> 1;
                recur @ [apndr@[1, expression], tl@2, tl@3])
}

{expression  * @ [1@3, - @ [1@2, last@1]] }

{LLL5 recur @ setup}
```

Figure A3.3 - Livermore Kernel No. 5 - First Version

Kernel 5: Tri-diagonal elimination, below diagonal (second version)

Kernel 5 can be rewritten as:

FORTRAN:

```
DO 5 I = 2, 1000
  X(I) = Z(I)*Y(I) - Z(I)*X(I-1)
5 CONTINUE
```

In the above form we see that all Z(I)*Y(I) can be done in parallel before the recurrence or:

```
DO 55 I = 2, 1000
  W(I) = Z(I)*Y(I)
55 CONTINUE
DO 5 I = 2, 1000
  X(I) = W(I) - Z(I)*X(I-1)
5 CONTINUE
```

The FP description would be:

Input object: < <x1> Y(1:n) Z(1:n) >

Output object: X(2:n)

```
{phase1 [1,                          # keep x1
        &* @ trans @ {tl@2, tl@3],     # generate W
        tl@3]                          # keep Z
}

{recur (null@2 -> 1;
                recur @ [apndr@[1, expression], tl@2, tl@3])
}

{expression   - @ [1@2, * @ [1@3, last@1]] }

{LLL5    recur @ phase1}
```

Figure A3.4 - Livermore Kernel No. 5 - Second Version

Kernel 6: Tri-diagonal elimination, above diagonal

FORTRAN:

```
DO 6 J = 3, 999, 3
  I = 1000 - J + 3
  X(I) =  X(I) - Z(I)*X(I+1)
  X(I-1) = X(I-1) - Z(I-1)*X(I)
  X(I-2) = X(I-2) - Z(I-2)*X(I-1)
6 CONTINUE
```

This loop can be rewritten as:

```
DO 6 J = 1, 999, 1
  I = 1000 - J + 1
  X(I) = X(I) - Z(I)*X(I+1)
6 CONTINUE
```

FP: input object: < X(1:n+1) Z(1:n) >
output object: X(1:n)

```
{setup   [tl@tlr@1, [last@1], tl@2]}

{recur (null@1 -> 2;
                  recur @ [tlr@1, apndl@[expression, 2], tlr@3])
}

{expression  - @ [last@1, * @ [1@2, 1@3]] }

{LLL6 recur @ setup}
```

Figure A3.5 - Livermore Kernel No. 6

Kernel 7: Equation of State Excerpt

FORTRAN:

```
DO 7 M = 1, 120
 X(M) = U(M) + R* (Z(M)+R*Y(M))+
    + S * ( U(M+3)+R*(U(M+2)+R*U(M+1))
7           + S * (U(M+6) + R*(U(M+5)+R*U(M+4))))
```

FP: input object: < Y(1:n), Z(1:n), U(1:n+6) >
output object: X(1:n)

```
{Ym 1}   # select Y(1:n)
{Zm 2}   # select Z(1:n)
{Um        tlr @ tlr @ tlr @ tlr @ tlr @ tlr @ 3 }  # select U(1:n)
{Umplus1  tlr @ tlr @ tlr @ tlr @ tlr @ tl  @ 3 }  # select U(2:n+1)
{Umplus2  tlr @ tlr @ tlr @ tlr @ tl  @ tl  @ 3 }  # select U(3:n+2)
{Umplus3  tlr @ tlr @ tlr @ tl  @ tl  @ tl  @ 3 }  # select U(4:n+3)
{Umplus4  tlr @ tlr @ tl  @ tl  @ tl  @ tl  @ 3 }  # select U(5:n+4)
{Umplus5  tlr @ tl  @ tl  @ tl  @ tl  @ tl  @ 3 }  # select U(6:n+5)
{Umplus6  tl  @ tl  @ tl  @ tl  @ tl  @ tl  @ 3 }  # select U(7:n+6)

# align operands
{align  [ Ym, Zm, Um, Umplus1, Umplus2, Umplus3, Umplus4, Umplus5, Umplus6 ] }

#  Expression will be divided in three clusters -- c1, c2, c3 --

{c3   * @ [S, + @ [9, * @ [R, + @ [8, * @ [R, 7 ] ] ] ] ] }

{c2   + @ [6, * @ [R, + @ [5, * @ [R, 4 ] ] ] ] }

{c1   + @ [3, * @ [R, + @ [2, * @ [R, 1 ] ] ] ] }

{Xm   + @ [c1, * @ [S, + @ [c2, c3]]] }   # the whole expression is c1 + S * (c2 + c3)

# LOOP:

{LLL7  & Xm   @  # execute all expressions
         trans  @  # put elements for each expression together
         align  }  # create and align all copies
```

Figure A3.6 - Livermore Kernel No. 7

Kernel 9: Integrate Predictors

```
DO 9 I = 1, 100
  PX(1,I) = BM28 * PX(13,I) + BM27 * PX(12,I) + BM26 * PX(11,I) +
        BM25 * PX(10,I) + BM24 * PX(9,I) + BM23 * PX(8,I) +
        BM22 * PX(7,I) +
        CO * (PX(5,I) + PX(6,I) + PX(3,I))
9 CONTINUE
```

FP: input object: < BM(22:28) PX(1:13, 1:100) >
output object: PX(1:100)
Note that row 1 is used to hold the output; rows 2 and 4 are not used in the calculation of the loop. Thus, we begin by throwing away rows 1, 2 and 4 of the matrix PX.
Output will be a vector (correspond to row 1 of PX in the FORTRAN loop).

```
# throw away unneeded rows of PX; align objects for expression calculation

{align [[1, tl @ tl @ tl @ tl @ tl @ tl @ 2], [3, 5, 6] @ 2]}

# output of align has two "elements"
# part1 to be applied to the 1st "element"; calculate all iterations
# for the part of expression that uses PX(7,*) to PX(13,*)

{part1    &(!+ @ &*)  @  trans @ &distl @ trans }

# to be applied to the 2nd "element"; calculate all iterations
# for the part of expression that uses PX(3,*), PX(5,*) and PX(6,*)

{part2    & (*@[CO, id])  @  &(!+)  @  trans }

{CO %1}   # constant CO

# LOOP:

{LLL9  &+    @
       trans  @
       [part1@1, part2@2]  @
       align
}
```

Figure A3.7 - Livermore Kernel No. 9

Kernel 10: Difference Predictors

FORTRAN:

```
DO 10 I = 1, 100
   AR = CX(5,I)
   BR = AR - PX(5,I)
   PX(5,I) = AR
   CR = BR - PX(6,I)
   PX(6,I) = BR
   AR = CR - PX(7,I)
   PX(7,I) = CR
   BR = AR - PX(8,I)
   PX(8,I) = AR
   CR = BR - PX(9,I)
   PX(9,I) = BR
   AR = CR - PX(10,I)
   PX(10,I) = CR
   BR = AR - PX(11,I)
   PX(11,I) = AR
   CR = BR - PX(12,I)
   PX(12,I) = BR
   PX(14,I) = CR - PX(13,I)
10 PX(13,I) = CR
```

This loop can be rewritten as (all iterations can be done in parallel):

```
DO 10 I = 1,100
   PX(14,I) = CX(5,I) - PX(5,I)  - PX(6,I)  - PX(7,I)  - PX(8,I)
            - PX(9,I) - PX(10,I) - PX(11,I) - PX(12,I) - PX(13,I)
   PX(13,I) = CX(5,I) - PX(5,I)  - PX(6,I)  - PX(7,I)  - PX(8,I)
            - PX(9,I) - PX(10,I) - PX(11,I) - PX(12,I)
   PX(12,I) = CX(5,I) - PX(5,I)  - PX(6,I)  - PX(7,I)  - PX(8,I)
            - PX(9,I) - PX(10,I) - PX(11,I)
   PX(11,I) = CX(5,I) - PX(5,I)  - PX(6,I)  - PX(7,I)  - PX(8,I)
            - PX(9,I) - PX(10,I)
   PX(10,I) = CX(5,I) - PX(5,I)  - PX(6,I)  - PX(7,I)  - PX(8,I)
            - PX(9,I)
   PX( 9,I) = CX(5,I) - PX(5,I)  - PX(6,I)  - PX(7,I)  - PX(8,I)
   PX( 8,I) = CX(5,I) - PX(5,I)  - PX(6,I)  - PX(7,I)
   PX( 7,I) = CX(5,I) - PX(5,I)  - PX(6,I)
   PX( 6,I) = CX(5,I) - PX(5,I)
10 PX( 5,I) = CX(5,I)
```

Figure A3.8 - Livermore Kernel No. 10

```
Kernel 10: Difference Predictors (continued)

FP: input object: < CX(1:n)  PX(5:14, 1:n) >
    output object: PX(5:14, 1:n)

# prepare input object to be operated:

{align   &apndl @ trans @ [1, trans@2] }

# FP description of each expression of the loop (10 expressions):

{expr  [1,
        -@[1,2],
        -@[-@[1,2],3],
        -@[-@[-@[1,2],3],4],
        -@[-@[-@[-@[1,2],3],4],5],
        -@[-@[-@[-@[-@[1,2],3],4],5],6],
        -@[-@[-@[-@[-@[-@[1,2],3],4],5],6],7],
        -@[-@[-@[-@[-@[-@[-@[1,2],3],4],5],6],7],8],
        -@[-@[-@[-@[-@[-@[-@[-@[1,2],3],4],5],6],7],8],9],
        -@[-@[-@[-@[-@[-@[-@[-@[-@[1,2],3],4],5],6],7],8],9],10]
       ]
}

# LOOP:

{LLL10  trans   @
        &expr   @
        align
}
```

Figure A3.8 (cont'd) - Livermore Kernel No. 10

Kernel 11: First Sum (first order linear recurrence)
(sum scan of APL)

```
X(1) = Y(1)
DO 11 K = 2, 1000
  X(K) = X(K-1) + Y(K)
11 CONTINUE
```

FP: input object : Y(1:n)
output object: X(1:n)

```
{setup [[1], tl]}

{recur (null@2 ->   1;
                    recur @ [apndr@[1, expression], tl@2])
}

{expression   + @ [last@1, 1@2] }

{LLL11   recur @ setup}
```

Figure A3.9 - Livermore Kernel No. 11

Kernel 12: First Difference (Vector subtraction)

```
DO 12 K = 1, 999
  X(K) = Y(K+1) - Y(K)
12 CONTINUE
```

FP: input object: Y(1:n)
output object: X(1:n-1)

```
{Yk   tlr}      {Ykplus1   tl}

{align   [Ykplus1, Yk]}

{LLL12   &- @ trans @ align }
```

Figure A3.10 - Livermore Kernel No. 12