SYMBOLIC STRUCTURAL TRANSFORMATIONS AND COMPILATION TECHNIQUES FOR FP

Jose Nagib Cotrim Arabe

October 1986
CSD-860025

SYMBOLIC STRUCTURAL TRANSFORMATIONS AND COMPILATION TECHNIQUES FOR FP [1]

José Nagib Cotrim Arabe [2]

Milos̆ D. Ercegovac

Computer Science Department, University of California, Los Angeles

Abstract

This work investigates a technique for the efficient execution of functional programs by reducing data replication and data movement. This is achieved by 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 basic algebraic relations, the system derives the structure of the result object without the need for the actual input object. This approach is a basis 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 some FP primitives at compile time; this process reduces the amount of data replication and data movement required by the original FP program. 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.

[1] This research was supported by ONR Contract N00014-83-K-0493, by the State of California MICRO-Rockwell Grant 157, and by CAPES, Ministry of Education, Brazil, under Contract 3906/81.

[2] Now at the Departamento de Ciência da Computação, Universidade Federal de Minas Gerais, 31271, Belo Horizonte, Brazil.

1. INTRODUCTION

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.

One proposal that has been made to overcome these problems is the use of a new class of programming languages. The class of *applicative* or *functional* programming languages has been 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. This works is based on the functional language FP described by Backus in [Back78].

In spite of their nice properties, functional languages still are not widely accepted in the real world of programming, one of the reasons being that they 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.

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 overhead;

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

d. in general, FP 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 often removes sources of inefficiency.

It has been recognized by applicative language researchers that one of the critical points in the execution efficiency of these languages is memory allocation and management. Most implementations handle very poorly regular data structures such as arrays and vectors. Since there is a big class of problems that manipulate arrays, it seems unlikely that functional languages will succeed if they do not handle well this class of problems. On the other hand, sources of inefficiency can be identified that, without regard to implementation, will be likely to contribute to slow down of 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.

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

Excessive data copying: It is well known that the introduction of redundancy often increases 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 speed. 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 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 [Back72] which is 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 article proposes 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 sections describe a new approach in the area of functional programming transformation with the objective of increased performance of execution.

2. SYMBOLIC STRUCTURAL TRANSFORMATIONS

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 section, 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 other works [Wadl81, Wadl84, Bell84, Augu84], 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 in Section 4, which solves an FP program structurally in order to generate efficient run-time environment for the actual execution of the FP program.

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 1.

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*

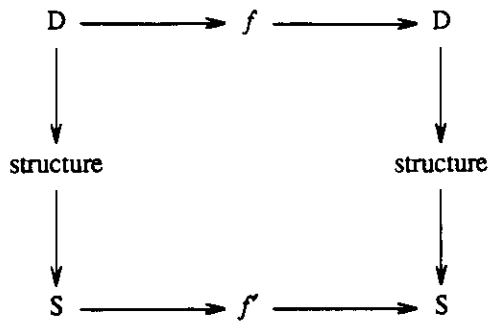


Figure 1 - Symbolic Structural Transformations

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 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.

2.2 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. Below we describe the basic *structural transformations* induced by some of the FP primitives. 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**.

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

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: $<s^{1:n}>$ and $n \ge 2 \rightarrow <s^{2:n}>$; Λ

distl: $<s, <t^{1:n}>> \rightarrow << s, t>^{1:n}>$; Λ

Restriction: Second element is a homogeneous sequence. **distr** has similar behavior.

trans: $< <s^{1:m}>^{1:n} > \rightarrow < <s^{1:n}>^{1:m} >$ $m, n \ge 1$; Λ

Restriction: Homogeneous sequences.

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

Restrictions: Homogeneous sequence and n even.

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

Restrictions: Homogeneous sequence and n even.

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

null: $<> \rightarrow$ **T; F**

length: $<s_1, s_2, \cdots, s_n> \rightarrow n$; $<> \rightarrow 0$; Λ

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

2.3 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 if $\sigma(x) \equiv \sigma(y)$ 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. For the class of structurally well-behaved functions and for homogeneous sequences, we have the following behavior for apply-to-all:

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

Constant

$$\%\, x : y \equiv x, \qquad \text{for all objects } x,\ y.$$

Conditional

Two types of conditionals are permitted. The first type acts 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:

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

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

3. 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 some 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.

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**) has similar behavior.

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

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

...

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

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 2.2. Therefore, there exists no change in the algebraic equations that describe the positions of regular structures when any of these FP primitives are applied.

4. MANIPULATION OF ALGEBRAIC EQUATIONS AND COMPILATION

In the previous sections, an algebra for symbolic structural evaluation of FP programs was defined. In this section, 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 2 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 [Back78]:

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

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

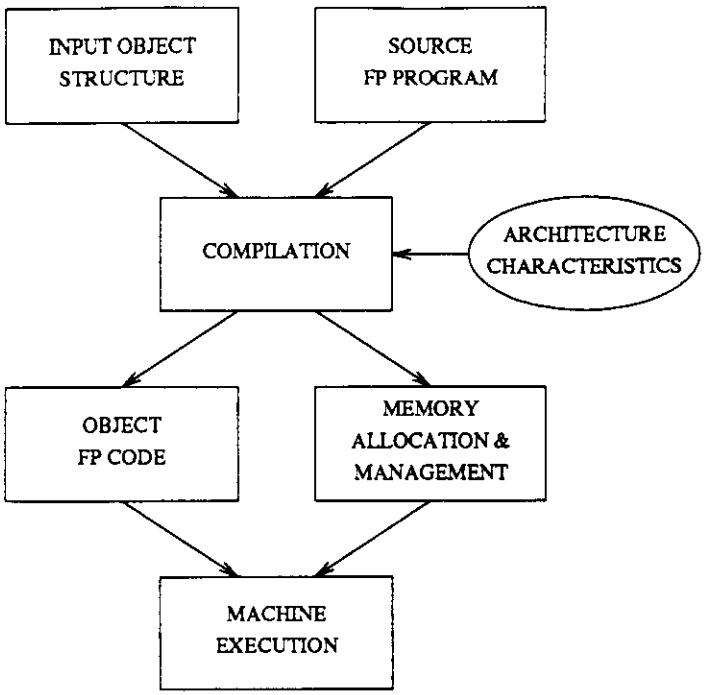


Figure 2 - Compilation: Proposed Approach for Optimization of FP Programs

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 inter-leaved 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 illustrates the matrices' elements positions for n=2, m=3, l=4):

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

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

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:

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

$$loc(b_{ij}) = nm+(i-1)+m(j-1),\ 1 \le i \le m,\ 1 \le j \le l \tag{2b}$$

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

$$loc(a_{ij}) = (m+ml)(i-1)+(j-1),\ 1 \le i \le n,\ 1 \le j \le m \tag{3a}$$

$$loc(b^{k}_{ij}) = m+(i-1)+m(j-1)+(m+ml)(k-1),\ 1 \le i \le m,\ 1 \le j \le l,\ 1 \le^{*} k \le n \tag{3b}$$

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 3; 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

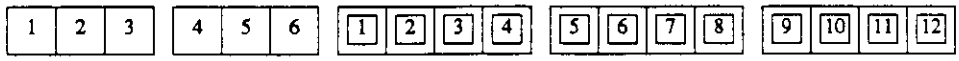
$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix}$$

A x B

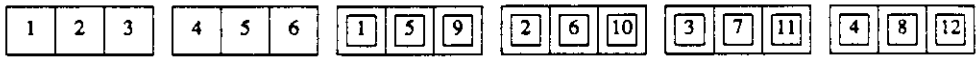
Elements of A are inside a 

Elements of B are inside a 

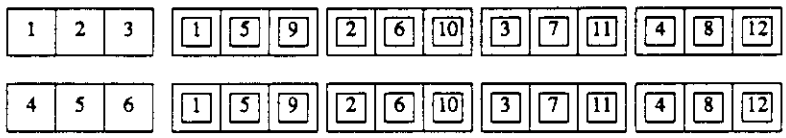
Initial positions:



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



After **distr**:



After **&distl**:

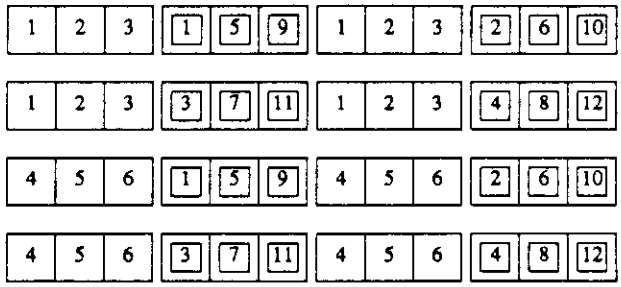
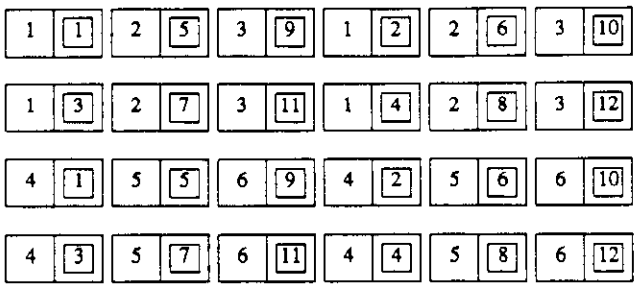


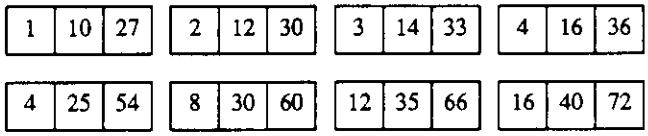
Figure 3 - MM Example for A(2 x 3) and B(3 x 4)

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

After **&&trans**:



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



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

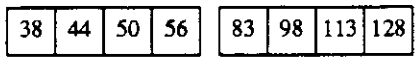


Figure 3 (cont'd) - MM Example for A(2 x 3) and B(3 x 4)

the following equations:

$$loc(a_{ij}^{k}) = 2ml(i-1)+(j-1)+2m(k-1),\ 1\le i\le n,\ 1\le j\le m,\ 1\le^{*} k\le l \tag{4a}$$

$$loc(b_{ij}^{k}) = m+(i-1)+2m(j-1)+2ml(k-1),\ 1\le i\le m,\ 1\le j\le l,\ 1\le^{*} k\le n \tag{4b}$$

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

$$loc(a_{ij}^{k}) = 2ml(i-1)+2(j-1)+2m(k-1),\ 1\le i\le n,\ 1\le j\le m,\ 1\le^{*} k\le l \tag{5a}$$

$$loc(b_{ij}^{k}) = 1+2(i-1)+2m(j-1)+2ml(k-1),\ 1\le i\le m,\ 1\le j\le l,\ 1\le^{*} k\le n \tag{5b}$$

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 5 and generate a

new object that we call C and that has its positions described by the following equations:

$$loc(c_{ijk}) = ml(i-1)+2(j-1)+(k-1),\ 1\le i\le n,\ 1\le j\le l,\ 1\le k\le m \tag{6}$$

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:

$$loc(d_{ij}) = l(i-1)+(j-1),\ 1\le i\le n,\ 1\le j\le l \tag{7}$$

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

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

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

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

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 (1) to (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 the first section;

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 (1) into equations (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.

5. PERFORMANCE EVALUATION

This section presents a summary of some performance evaluation results done on the approach described in the previous sections. For a more detailed description, refer to [Arab86]. We investigated the effects of the structural transformation techniques at compile time on a conventional uniprocessor model. A comparison was made between two approaches: the traditional interpreted mode of execution of FP programs versus the compilation mode proposed in this work.

We assumed that both the compiler and the interpreter run in a conventional von Neumann architecture. For the model, we also assumed a non-interleaved linear memory; one-word-at-a-time transfer between processor and memory; one memory word capable of storing any atom; and a string reduction execution mode, which implies that distinct copies of actual objects are generated for each function application.

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:

a. *Memory Requirements*: We use the amount of data memory required for the execution of an FP program, without reuse of storage, as the measure of data replication for comparison between the two approaches. M_{int} is the memory requirements for the interpretation case and M_{comp} for the compilation case. We also define $R_M = M_{int}/M_{comp}$ to observe the ratio between the two cases.

b. *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. We denote BT_{int} the bus traffic for the interpreted case and BT_{comp} the bus traffic for the compiled

case. Finally, we define the ratio $R_{BT} = BT_{ini}/BT_{comp}$.

We have examined some characteristic FP programs under the measurements defined above: the matrix multiplication program – it is a typical representative of an arithmetic intensive problem; another vector processing problem, but with an additional characteristic: the Fast-Fourier Transform is a recursive program; some purely structural programs (descriptions of interconnection patterns); and, finally, an associative searching problem, which has the characteristic of not being numeric intensive.

Table 1 summarizes the memory requirements improvements for the FP programs analyzed. 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 1 - Summary of Memory Requirements Results

The summary of results for the bus traffic is shown in Table 2 – 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 2 - Summary of Bus Traffic Results

6. CONCLUDING REMARKS

This paper has presented 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. We also showed 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 in a given memory organization. The manipulation of these algebraic equations allows the optimization of FP programs at compile time.

REFERENCES

[Arab86] Arabe, J.N.C., "Compiler Considerations and Run-Time Storage Management for a Functional Programming System," Technical Report No. CSD-860041 , Ph.D. Dissertation, Computer Science Department, University of California, Los Angeles (August 1986).

[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).

[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).

[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).

[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).

[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).

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