McGill University
School of Computer Science

An Efficient Monolithic Array Constructor

ACAPS Technical Memo 19 (Revised)
September 13, 1990

G. R. Gao
Robert Kim Yates
Advanced Computer Architecture and Program Structures Group

Jack B. Dennis
Laboratory for Computer Science
Massachusetts Institute of Technology

Lenore Restifo Mullin
Centre de Recherche Informatique de Montréal

3480 University St • Montreal • Canada • H3A 2A7

Abstract

Despite their importance in scientific computation, arrays have posed a serious persistent challenge to the efficient implementation of functional languages. In this paper, we propose a new monolithic array constructor which provides a solution to two major problems arising in the implementation of monolithic arrays: (1) the overhead of scheduling and synchronization for recursively defined arrays; (2) the copying of the many intermediate arrays during array construction. Our array constructor allows an index domain to be constructed from a set of rectangular *index regions* and combinations of these. An ordered sequence of regions may be specified in a fashion which can be checked by a compiler using simple subscript analysis techniques. An array is also allowed to be monotonically defined in an iterative fashion without using incremental array update operations. Program examples arising in scientific computation are used to illustrate the principles and advantages of the new constructor.

Contents

1 Introduction: Arrays in Functional Languages **1**

2 Monolithic Array Constructors **3**

3 A New Monolithic Array Constructor **4**

4 Recursive Array Construction **6**

5 A Solution to the Wavefront Problem **10**

6 Compile-time Checking **11**

7 Subscript Analysis **11**

8 The LU-decomposition Problem **14**

9 Monotonic Definition of an Iteration Variable **15**

10 Future Work **18**

A Syntax of the Monolithic Array Constructor **19**

List of Figures

1 Syntax for the basic monolithic array constructor 5

2 NodeReflect written with new monolithic constructor. 7

3 NodeReflect written in Sisal. 8

4 LU-Decomposition using Gaussian elimination 15

5 LU-Decomposition written in Id-Nouveau 16

1 Introduction: Arrays in Functional Languages

Issues concerning the efficient implementation of arrays have received little attention from the designers of functional programming languages. From the beginnings of interest in functional programming to the present, most published work has developed within the framework of symbolic applications and interactive computing [9]. This is surprising in view of the intense current interest in exploiting the power of parallel computers for high performance scientific and engineering computation, since the absence of side effects in functional languages facilitates the identification of program parts that may be executed concurrently. Notable exceptions have been APL [23] (although not truly a functional language), and the work on array theory [28, 29]. Two functional programming languages, Val and Sisal [2, 27], have been designed specifically for scientific computation and include expression forms for the construction of multidimensional arrays. Id-Nouveau has also paid considerable attention to supporting arrays for scientific computation [31].

Two cases of array construction are important. In the first case, each element of the array has a value obtained using only values available before construction of the array begins, and elements may be computed concurrently without conflict or hazard. This case is supported by many languages, e.g., the **forall** of Val and the parallel **for** expression in Sisal. These constructs are *monolithic* in the sense that their elements are defined "all at once" at the moment the array value is created [3].

The second case occurs if the expression for the value of an array element contains references to other elements of the array being constructed. This case often arises in scientific computation when the array elements are determined by a mathematical recurrence – each array element may depend on several elements of the same array. In Val and Sisal this requires use of the array update operation and leads to the copying of array values in unoptimized machine code.

An array update operation (also sometimes called "append") `A[i:v]` generates a new array which agrees everywhere with the old array A except at the i^{th} position, where it has the value v. The creation of the new array leaves the old array intact, thereby preserving referential transparency. However, such *incremental* array update operations, when implemented literally, incur large costs in time and space due to copying.

There have been many attempts to eliminate or reduce incremental array copying in functional languages. In early proposals it was suggested that arrays might be implemented as trees and copying avoided by proper sharing of subarrays [1, 13]. But implementing arrays as trees sacrifices the efficient accessing of elements supported by familiar memory system architectures. Another approach combines static analysis with runtime reference counting to eliminate some copying and memory management operations [7, 12, 16, 19, 34]. It is not clear how far these methods can be extended. An alternative is to refine the programming

language so that the function of the append operation is replaced by constructs more amenable to analysis by a compiler.

In recent years, monolithic arrays have been included in several functional languages, including FEL [24], Alfl [15], Id-Nouveau [32] and Haskell [20]. Related proposals can also be found in [36] and [37]. Hudak [17] has summarized the two major difficulties confronting the implementors of languages that include these array constructors: (1) the overhead of scheduling and synchronization for recursively defined arrays; and (2) the copying of the many intermediate arrays during array construction.

More powerful array operations are being considered in extending conventional programming languages We lack the space to give even a short survey, and we refer the readers to Fortran 8X as a representative example of a general trend in this direction [21]. In the language FIDIL [14], domains and maps play a central role and constitute an extension of the array types found in other programming languages. A FIDIL domain corresponds to an index set similar to our index region and maps are functions which can monolithically construct an array based its index domain. A rich set of operations on domains and maps is included in the language. FIDIL's objective is to reduce the semantic distance between abstract descriptions of finite difference and particle methods and programs that implement them, making abstract mathematical descriptions into programs. To this effect, FIDIL encourages programmers to build arbitrary complex array object in single expressions, using its powerful array operations.

The objective of this paper is to study issues in funtional language design that influence our ability to achieve efficient implementation of monolithic arrays intended for scientific and engineering computation. Careful attention to language design can make the construction of efficient code much more straightforward. Toward this end, we propose a new monolithic array constructor which allows an index domain to be constructed from a set of rectangular *index regions* and combinations of these. An ordered sequence of regions may be specified in a fashion which can be checked by a compiler using simple subscript analysis techniques. An array is also allowed to be monotonically defined in an iterative fashion without using incremental array update operations. Program examples arising in scientific computation are used to illustrate the principles and advantages of the new constructor.

In section 2 we review monolithic array constructors as they have been defined in existing functional programming languages, and illustrate some deficiencies of current languages for scientific computation. Section 3 introduces our basic monolithic array constructor for non-recursive array definitions. In section 4 the new array constructor is extended to recursive array definitions and illustrated for the wavefront computation. Sections 6 and 7 treat subscript analysis. The problem of minimizing copying, using LU decomposition as an example, is illustrated in section 8; a new approach to copy elimination is presented in section 9. The paper concludes with comments on future work.

2 Monolithic Array Constructors

It is the recursive array definitions that provide the greatest challenge for the design of a monolithic array constructor. One would like to ensure that: (1) arbitrary dependences can be expressed; (2) "in place" construction may be used to minimize storage allocation needs; (3) possible concurrency in the evaluation of array elements can be exploited. Two interesting proposals for meeting these objectives exist in the facilities provided in Id and in the newly proposed language Haskell.

The intention in Id is that array definitions having arbitrary recursive dependences be written using the *I-structure* mechanism of Id. This mechanism alows for explicit allocation of storage for an array with all elements undefined, and supports write and read operations on the array elements. It is invalid to write the same element twice (an error that can only be detected during program execution). Any reads of an array element are delayed until the element becomes defined. The general implementation of I-structures requires a special memory organization and a resource scheduling mechanism that may be invoked for each element of the constructed array. To avoid the penalties of using I-structures, the Id program could be analyzed by a compiler to recognize the important simple forms for which more efficient code can be constructed. The problems of doing this are similar to those for the monolithic array constructor in Haskell, which we introduce next.

The functional programming language Haskell [20] has been proposed as a standard functional language for a broad range of applications. Haskell provides an expression form in which an array is specified by a list of pairs of the form $(index, value)$, where *index* is a tuple of integers, one for each dimension of the array being constructed, and *value* is an object of element type. The list is written as a concatenation of one or more *list comprehensions* [33]. Such an expression defines an array monolithically and is known as an *array comprehension.* As an example, the wavefront computation may be written as follows [17]:

```
A = array ((1,m),(1,n)) [ ((1,1),1) ]
                        ++ [ ((i,1),1) | i <- [2..m] ]
                        ++ [ ((1,j),1) | j <- [2..n] ]
                        ++ [ ((i,j), A!(i-1,j) + A!(i,j-1) + A!(i-1,j-1) )
                                     | i <- [2..m], j <- [2..n]]
```

The right hand side of this definition consists of the keyword `array`, a bounds expression `((1,m),(1,n))` that specifies the rank and index ranges of the constructed array, and a group of four list comprehensions combined with the concatenate operator "`++`". The first list contains a single element that defines the corner element to be one. The next two lists define the remaining elements of the first column and the first row of the array. The last list is specified using the Zermelo-Frankel set abstraction notation to express the dependence of the

interior array elements on their neighbors to the north, west and northwest. The exclamation point "`!`" is the subscripting operator.

In this way the Haskell program uses list comprehensions to specify four different regions which cover the index domain of A. In the wavefront example, the list comprehension for the interior region is created by two generators, one for i and one for j, producing index pairs for all points (i,j) not part of the north and west boundaries. To appreciate the semantics of an array comprehension, it is important to know that the order in which array elements are specified in the list *does not* specify the order in which element values are to be computed; the list is only a device for specifying the *values* of array elements. It is the responsibility of the implementation (the compiler and runtime system) to use an order of evaluation that honors the dependences expressed in the array comprehension. For a parallel computer it is also the job of the implementation to determine which elements may be safely evaluated concurrently. In the wavefront example, it is the sets of elements that make up the secondary diagonals of the array.

In many scientific computations an array may be recursively defined, but the pattern of self reference is regular, as in a linear recurrence or the wavefront computation. In these cases it is important that the inefficiencies of the general implementation of the Id and Haskell array constructors be avoided. Compile-time analysis can help if the regularities in the dependence relation can be identified and used to partition array elements into groups that can be safely evaluated in parallel. As will be shown in section 6, array constructors can be designed to make this process straightforward.

3 A New Monolithic Array Constructor

In this section we introduce the basic form of our monolithic array constructor. This form applies to arrays defined without self-dependence. The extension to recursive array definition is made in section 4.

We consider an array to be a function whose domain is a set of tuples of integers. Such a domain may be thought of as a rectangular subset of n-dimensional space. We call this rectangular region the *index domain* of the array. Our array constructor lets the programmer specify an array by giving separate rules for several *index regions* within the index domain. Each index region may be specified as a union of rectangular regions and points. Although this could be expressed using nested conditional expressions that test index values, the use of expressions guarded by region specifications yields a more natural and compact format. This is done by allowing the body of the array constructor to consist of a set of pairs, each consisting of a list of *region specifications* and an array element expression.

The syntax of the basic array constructor is given in Figure 1. An n-dimensional rectangular index region R can be denoted by a pair `[`*shape*`,`*lowbd*`]`, where both *shape* and *lowbd*

```
mono-array-constructor ::= for index-list in domain-spec construct [id]
                               mono-body
                           end for
domain-spec            ::= [ shape , lowbd ]
mono-body              ::= region-constructor [ ; mono-body ]
                         | region-constructor [ ! mono-body ]
region-constructor     ::= region-spec : expression
region-spec            ::= [ shape , lowbd ]
                         | point
                         | otherwise
shape                  ::= ( int-expr-list )
lowbd                  ::= point
point                  ::= ( int-expr-list )
int-expr-list          ::= int-expr [ , int-expr-list ]
int-expr               :=  integer expressions evaluated prior to array constructor
expression             :=  arbitrary expression of array element type
```

Figure 1: Syntax for the basic monolithic array constructor

are ordered n-tuples of integers. This format is adapted from [29]. Each element of the *shape* tuple is the length of the corresponding dimension of the specified region; the *lowbd* tuple specifies the origin of the region. For example, a 10-by-10 two-dimensional index region with lower bound at (0,0) is represented by `[(10,10),(0,0)]`.

The *domain-spec* defines the index region as a shape/lower bound pair. The *mono-body* specifies a list of region constructors separated by the parallel or sequential composition operator "`;`" or "`!`". A region constructor may be a *region-spec* followed by an expression for computing array elements in that region. Each region-spec defines a subspace within the index domain by specifying a rectangular subspace, or by giving a list of points.

The semantics of the array constructor is strict: the construct fails to returns an array value if the computation of any element fails to complete. This decision is based on the observation that strict semantics is satisfactory for most arrays constructed in scientific computation. The difference between strict and nonstrict semantics is important. Some expressions that terminate with valid results under nonstrict semantics will fail to terminate under strict semantics. There is a debatable question as to whether programs for which this distinction exists are "good" programs. Another difference is that greater parallelism may be exploitable under a nonstrict semantics because, for example, the consumer of an array value may begin processing elements before the producer has given values to all elements. However, implementations honoring strict semantics can also use elements before all have been computed.

The convenience of the array constructor for real scientific computation may be illustrated by an example from the SIMPLE benchmark – the NodeReflect routine[8]. In this routine, the boundary elements of a data array are defined by different formulas for each part of the boundary.

As shown in Figure 2, there are 13 pieces that must be separately defined. The complexity of assembling the result array is illustrated by the Sisal code for NodeReflect given in Figure 3. In this form of the code the 13 components are constructed as separate arrays and concatenated, horizontally and then vertically, to form the result. Though current Sisal compilers are actually able to determine that this code may be optimized to the construction of a single array without any copying, this style of programming is very awkward. Alternatively, it could be written in Sisal as a single parallel **for** expression in which the body is an elaborate nested conditional that tests which of the 13 regions apply to the current index values and selects the proper formula. Although the tests on the indices may be optimized away, the new code would still be awkward.

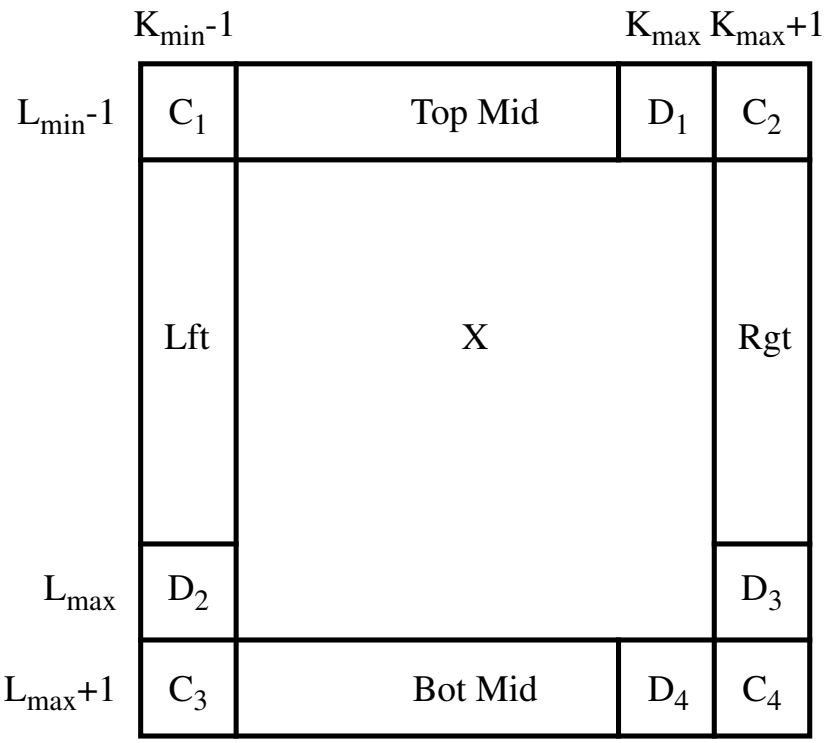
Conventional languages with their side effects can use assignment statements to write the values into memory slots at will, thus avoiding the problem of having to combine various bits and pieces explicitly into a coherent whole. But by using the new monolithic array constructor with a partitioned index domain as shown in Figure 2, we can keep the important advantage of referential transparency found in purely functional languages, yet avoid the problems of incremental construction. The new code uses no concatenations, no copying, and no conditionals.

The level of performance that is achievable for programs using the monolithic array constructor will depend on characteristics of the target machine and on the information available at compile time. If the parameter values that determine the position and shape of the index domain and the region specifications are not known to the compiler, then many decisions about storage allocation and process partitioning and scheduling must be deferred to execution time. If parameter values are known at compile time, the compiler can check that the specified regions do not overlap and that they collectively fill up the index domain.

Index regions of other shapes such as diagonals or hyperplanes are also useful in scientific computation. We will illustrate their specification later in the paper. The next section extends the constructor to recursively defined arrays.

4 Recursive Array Construction

An array is *recursively defined* if some elements of the array depend on other elements. For example, the array A in the wavefront example introduced in section refsec-mono is recursively defined: each element of A depends on its north, west and northwest neighbours (except the elements of the top and left boundaries).




```
for i,j in [(Lmax-Lmin+1,Kmax-Kmin+1),(Lmin-1,Kmin-1)] construct
  [(1,1),(Lmin-1,Kmin-1)] : 0.0;                                    % C1
  [(1,1),(Lmin-1,Kmax+1)] : 0.0;                                    % C2
  [(1,1),(Lmax+1,Kmin-1)] : 0.0;                                    % C3
  [(1,1),(Lmax+1,Kmax+1)] : 0.0;                                    % C4
  [(1,1),(Lmin-1,Kmax)]
       : Projct( X[Lmin,Kmax], X[Lmin,Kmax-1], X[Lmin+1,Kmax]); % D1
  [(1,1),(Lmax,Kmin-1)]
       : Projct( X[Lmax,Kmin], X[Lmax-1,Kmin], X[Lmax,Kmin+1]); % D2
  [(1,1),(Lmax+1,Kmax)]
       : Projct( X[Lmax,Kmax], X[Lmax,Kmax-1], X[Lmax-1,Kmax]); % D3
  [(1,1),(Lmax+1,Kmax)]
       : Projct( X[Lmax,Kmax], X[Lmax,Kmax-1], X[Lmax-1,Kmax]); % D4
  [(1,Kmax-Kmin),(Lmin-1,Kmin)]
       : Projct( X[Lmin,j], X[Lmin,j+1], X[Lmin+1,j] );         % TopMid
  [(1,Kmax-Kmin),(Lmax+1,Kmin)]
       : Projct( X[Lmax,j], X[Lmax,j+1], X[Lmax-1,j] );         % BotMid
  [(Lmax-Lmin,1),(Lmin,Kmin-1)]
       : Projct( X[L,Kmin], X[L+1,Kmin], X[L,Kmin+1] );         % Lft
  [(Lmax-Lmin,1),(Lmin,Kmax+1)]
       : Projct( X[L,Kmax], X[L+1,Kmax], X[L,Kmax-1] );         % Rgt
  otherwise               : X[i,j]                                  % interior
end for
```

Figure 2: NodeReflect written with new monolithic constructor.

```
function NodeReflect(X:TwoDim; Kmin,Kmax,Lmin,Lmax:integer returns TwoDim)
let
  C1,C2,C3,C4 := 0.0, 0.0, 0.0, 0.0;
  D1     := Projct( X[Kmin,Lmax], X[Kmin,Lmax-1], X[Kmin+1,Lmax] );
  D2     := Projct( X[Kmax,Lmax], X[Kmax-1,Lmax], X[Kmax,Lmax-1] );
  D3     := Projct( X[Kmax,Lmin], X[Kmax-1,Lmin], X[Kmax,Lmin-1] );
  D4     := Projct( X[Kmax,Lmax], X[Kmax,Lmax-1], X[Kmax-1,Lmax] );
  TopMid := for L in Lmin,Lmax-1
            returns array of Projct( X[Kmin,L], X[Kmin,L+1], X[Kmin+1,L] )
            end for;
  BotMid := for L in Lmin,Lmax-1
            returns array of Projct( X[Kmax,L], X[Kmax,L+1], X[Kmax-1,L] )
            end for;
  MidRows:= for K in Kmin,Kmax-1
              Lft := Projct( X[K,Lmin], X[K+1,Lmin], X[K,Lmin+1] );
              Rgt := Projct( X[K,Lmax], X[K+1,Lmax], X[K,Lmax-1] );
              Row := array [Lmin-1:Lft] || X[K] || array [1:Rgt]
            returns array of Row
            end for;
  TopRow := array [Lmin-1:C1] || TopMid  || array [1:D1,C2]
  LowRow := array [Lmin-1:D2] || X[Kmax] || array [1:D3];
  BotRow := array [Lmin-1:C3] || BotMid  || array [1:D4,C4]
in   array [Kmin-1:TopRow]
  || MidRows
  || array [1:LowRow]
  || array [1:BotRow]
end let
end function
```

Figure 3: NodeReflect written in Sisal.

The data dependences between elements of a recursively defined array may put constraints on the order of their evaluation. In the Haskell array comprehension, the order of evaluation of array elements is not explicitly specified by the programmer. If a compiler does not identify the dependences, a general (and inefficient) implementation scheme must be used.

Two strategies that may be used to avoid such overhead in implementations of monolithic array constructors are:

Strategy 1: Have the compiler attempt to identify dependences without assistance from the programmer. Use optimized code for common regularities that can be recognized. Use a general implementation scheme as the default.

Strategy 2: Provide means in the language to explicitly indicate evaluation order where this information may help a compiler to construct a more efficient implementation.

Researchers have been working on both compile-time and runtime methods based on strategy 1 [3, 4]. Subscript analysis, originally developed for vectorizing/parallelizing compilers of imperative languages, can be adapted to find some efficient and safe evaluation order.

To pursue strategy 2 for recursive array definition, we extend our basic monolithic array constructor with features which allow programmers to express explicitly sequential evaluation orderings wherever they are needed both within and between regions. We add a *sequence-constructor* which specifies a region of the array to be recursively defined:

region-constructor	::=	*basic-constructor*
	|	*sequence-constructor*
basic-constructor	::=	*region-spec* : *expression*
sequence-constructor	::=	`ordered by` *id* `from` *range* *region-constructor*

Let us illustrate the construct by an example. The following program constructs an array where each column is computed by a first-order linear recurrence.

```
for i,j in [(n,n),(1,1)] construct AT
    [(1,n),(1,1)] : 1 !                          % Init top row first,
ordered by i from 2..n                           % then iterate over
    [(1,n),(i,1)] : B[i,j] + AT[i-1,j] + C[i,j]  % the lower n-1 rows.
end for
```

First, we note that an internal name, AT, for the recursively defined array is introduced after the keyword **construct**. The semantics allows AT to be referenced inside the scope of

the monolithic constructor, making it possible to reference the elements of the array before it is completely constructed.

The above program partitions the index region into rows. Following initialization of the first row, the `ordered` clause specifies that the remaining rows be evaluated in ascending sequence, row 2, row 3, etc., up to row n. Because no other ordering is specified, evaluation of array elements within each row can, by default, be performed in parallel.

The order specified between the regions will be checked by a compiler to confirm that the array elements are in fact defined before they are used in the array construction. If such checks fail, appropriate measures must be taken to warn the user. In may cases, our array constructor makes such checks straightforward. We discuss the issue further in the next section.

5 A Solution to the Wavefront Problem

So far, we have considered simple partitions of the index domain such as rows or columns. Other partitions may be more beneficial depending on the computation to be performed. The wavefront program discussed earlier is a good example where maximum parallelism would be achieved by evaluating the upper and left border elements all in parallel, then processing each of the secondary diagonals in sequence starting from the upper left corner (elements within each diagonal can be evaluated in parallel). There has been interesting work done on automatic detection and parallelization of wavefronts [6, 22, 25]. Our monolithic array constructor can express such an evaluation scheme:

```
for i,j in [(n,n),(1,1)] construct AT
    [(1,n),(1,1)]               : 1;  % initialize top row
    [(n-1,1),(2,1)]             : 1!  % initialize left column
ordered by k from 4..2*n              % fill the interior by diagonals
    [i+j=k] & [(n-1,n-1),(2,2)] : AT[i-1,j] + AT[i,j-1] + AT[i-1,j-1]
end for
```

In this code the entire index domain is partitioned such that the first row and first column are initialized simultaneously. The rest of the index domain is partitioned into $2n-3$ secondary diagonals. Note that the diagonals are specified by a new kind of *region-spec*, i.e. the equation $i+j=k$ of a line in the index space. More precisely, the region is implicitly defined as the intersection ("`&`") of the straight lines $i+j=k$ and the enclosing rectangular region `[(n-1,n-1),(2,2)]`, for $k = 4..2n-1$.

The construction of the diagonals is ordered explicitly by the ascending range given for k, and proceeds from top-left to bottom-right. The elements in any one diagonal may, by default,

be evaluated in parallel. It is easy to check that the specified evaluation order is consistent with the data dependences. We only need to check the following condition for consistency: all references to the result array must lie in a region already defined, i.e., on the boundary or on one side of the diagonal line $i + j = k$. In this particular example, the subscripts of `AT[i-1,j]`, `AT[i,j-1]`, and `AT[i-1,j-1]` all do in fact meet this straightforward test.

6 Compile-time Checking

First, we would like to contrast our subscript analysis to the traditional approach taken by parallelizing compilers for von Neumann languages such as Fortran and to the approach an optimizing compiler of a functional language such as Haskell may take.

Programs written in imperative languages *overspecify* the evaluation order of array elements: the sequential execution of statements yields a sequence of assignments to array elements. A parallelizing/vectorizing compiler performs array subscript analysis to detect the true dependences so the program may be transformed to exploit parallelism. The transformation must be *safe*, i.e., one that preserves all true dependences. In order to assure safety, a compiler must decide whether or not a given set of linear Diophantine equations in N variables has an integer solution in a certain index region [5]. No efficient algorithm has been found to derive a necessary and sufficient condition for such a test in general. To ensure safeness, a compiler must be conservative if the tests fail to exclude the possibility of dependence.

At the other extreme, a monolithic array constructor such as the Haskell array comprehension *underspecifies* the evaluation order. In fact no evaluation ordering is given at all. An optimizing compiler needs to perform subscript analysis to determine a correct evaluation order, if one exists.

The monolithic array constructor described in this paper allows the programmer to introduce explicit sequential ordering between regions of the array. The default "ordering" is parallel. In contrast to the two types of compilers mentioned above, the role of our compiler is to check that the programmer-specified order is consistent with the data dependences of the array definition. In other word, the compiler must check that the specified order of regions ensures that array elements are defined before they are used. Unlike a parallelizing compiler for Fortran, our compiler has no "first-order" obligation to parallelize the sequential order explicitly introduced by the programmer. Unlike an optimizing compiler for Haskell, our compiler does not need to discover a safe sequential order.

7 Subscript Analysis

In a recursive array definition, an array element may be computed from other elements of the same array. In using the proposed array constructor, it is the programmer's responsibility

to specify a correct evaluation order. The compiler should detect data dependences that are inconsistent with the specified order and report these as errors. In our case, the test to be made is that recursive references must fall only in regions that are earlier in the specified evaluation order. This amounts to making sure that a read of an array element is performed only after it has been defined (or "written", in the parlance of imperative languages).

Let us consider the general case:

```
for (x_1, ..., x_n) in region-spec construct A
    ... (initialization here) ...
    ordered by k from range
        [k = c_1 x_1 + c_2 x_2 + ... + c_n x_n]                          % a hyperplane
            : ... A[f_1(x_1, ..., x_n), f_2(x_1, ..., x_n), ..., f_n(x_1, ..., x_n)] ...   % recursive ref
end for
```

where the user specifies a sequence of hyperplanes H_k ($\forall k \in range$) in an n-dimensional index region and the k-th hyperplane H_k is defined using a linear function with n indices $x_1, x_2 ... x_n$:

$$c_1 x_1 + c_2 x_2 + ... + c_n x_n \quad = \quad k \tag{1}$$

We assume that $c_1, c_2 ... c_n$ are integer constants known at compile time. Using the monolithic array constructor, the elements of an array A lying on the hyperplane H are computed, with possible references to other elements of array A in the form

$$A[f_1(x_1, ..., x_n), f_2(x_1, ..., x_n), ..., f_n(x_1, ..., x_n)]$$

where the f_i are linear, i.e.

$$f_i(x_1, ..., x_n) \quad = \quad a_{i,0} + a_{i,1} x_1 + a_{i,2} x_2 + ... + a_{i,n} x_n \tag{2}$$

and $a_{i,0}, a_{i,1} ... a_{i,n}$ are constants. The problem is to check that for all k, the following holds:

$$c_1 f_1(x_1, ..., x_n) + c_2 f_2(x_1, ..., x_n) + ... + c_n f_n(x_1, ..., x_n) \quad < \quad k \tag{3}$$

under the conditions

$$l_i \leq x_i \leq h_i \qquad \forall i \in 1..n \tag{4}$$

where l_i, h_i are the bounds on the i-th dimensions of the index region.

For example, in the wavefront program, the hyperplanes are just the secondary diagonals of the 2-dimensional index region

$$c_i = 1, \quad \forall i \in 1..2$$

Consider, for instance, the array reference `A[i-1,j-1]`. We have

$$\begin{aligned} a_{1,1} = \quad a_{2,2} \quad &= 1 \\ a_{1,0} = \quad a_{2,0} \quad &= -1 \\ \qquad\quad a_{i,j} \quad &= 0 \qquad \text{otherwise} \end{aligned}$$

The consistency check is trivial in this case.

For the general case, the test expressed by inequality (3) can be rewritten in the form:

$$\sum_{i=1}^{n} b_i x_i - k \quad < \quad -b_0 \tag{5}$$

where

$$b_i \quad = \quad \sum_{j=1}^{n} c_j a_{j,i} \qquad\qquad \forall i \in 0..n \tag{6}$$

Substituting the left hand side of (1) for k in (5):

$$\sum_{i=1}^{n} b_i x_i - \sum_{i=1}^{n} c_i x_i \quad < \quad -b_0 \tag{7}$$

then collecting terms we end up with

$$\sum_{i=1}^{n} d_i x_i \quad < \quad -b_0 \tag{8}$$

where

$$d_i \quad = \quad b_i - c_i \tag{9}$$

Note that the $a_{j,i}$, b_i, c_i and d_i are all compile-time constants. Therefore, our problem of testing whether inequality (8) holds under condition (4) reduces to the standard problem of computing the bounds of a linear function in a trapezoidal region [5](Chapter 4). Both closed form solutions and reasonable algorithms exist, and the test can be performed without having to solve a system of linear Diophantine equations (what a relief!). In fact, for the most common cases the test is really trivial, as we have seen in the wavefront example. If the test succeeds, we are assured that the specified order is consistent with the data dependences. If the test fails, the program is erroneous and the compiler should report back to the programmer for debugging purposes.

The analysis described above can be put to even better use. Often an iterative relaxation algorithm will read old values from in front of the wavefront as well as new values from behind the wavefront, e.g.:

$$A^{t+1}[i,j] = (A^{t+1}[i-1,j] + A^{t+1}[i,j-1] + A^{t}[i,j+1] + A^{t}[i+1,j])/4$$

Under such conditions the the preceding analysis will verify that an implementation may safely overwrite A^t with A^{t+1}.

The analysis method we have presented assumes that array references in the body expression have only linear subscript expressions. From previous parallelization work for conventional languages such as Fortran [26], a substantial percentage of scientific programs appear to satisfy this requirement. Finally, if the body expression contains function calls, the compiler may need to perform analysis of the function invocation. We believe such analysis will be much simpler than a general interprocedural analysis for conventional languages due to the absence of side effects.

8 The LU-decomposition Problem

One problem that has attracted a lot of attention as a test case for array facilities in functional languages is LU-decomposition of matrices.[1] In this section we present our solution to the problem and illustrate its advantges.

LU decomposition is used to solve systems of linear equations of the form:

$$\begin{bmatrix} a_{1,1} & \cdots & a_{1,n} \\ & \cdots & \\ a_{n,1} & \cdots & a_{n,n} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ \vdots \\ b_n \end{bmatrix}$$

This can be expressed by the equation $Ax = b$, where A is a square matrix. In LU-decomposition A is decomposed into two submatrices:

- L: A lower triangular matrix which excludes the diagonal
  (more precisely, all diagonals of LU are equal to 1 and are omitted);
- U: An upper triangular matrix which includes the diagonal.

$$\begin{bmatrix} & U \\ L & \end{bmatrix}$$

One solution method for LU-decomposition is to use Gaussian elimination [35]. To simplify the discussion, we assume that no pivoting is performed. The structure of the algorithm is illustrated in Figure 4.

[1] LU-decomposition was suggested as a revealing exercise by Arvind during the IFIP WG 2.8 meeting in Mystic, Connecticut in May, 1989.

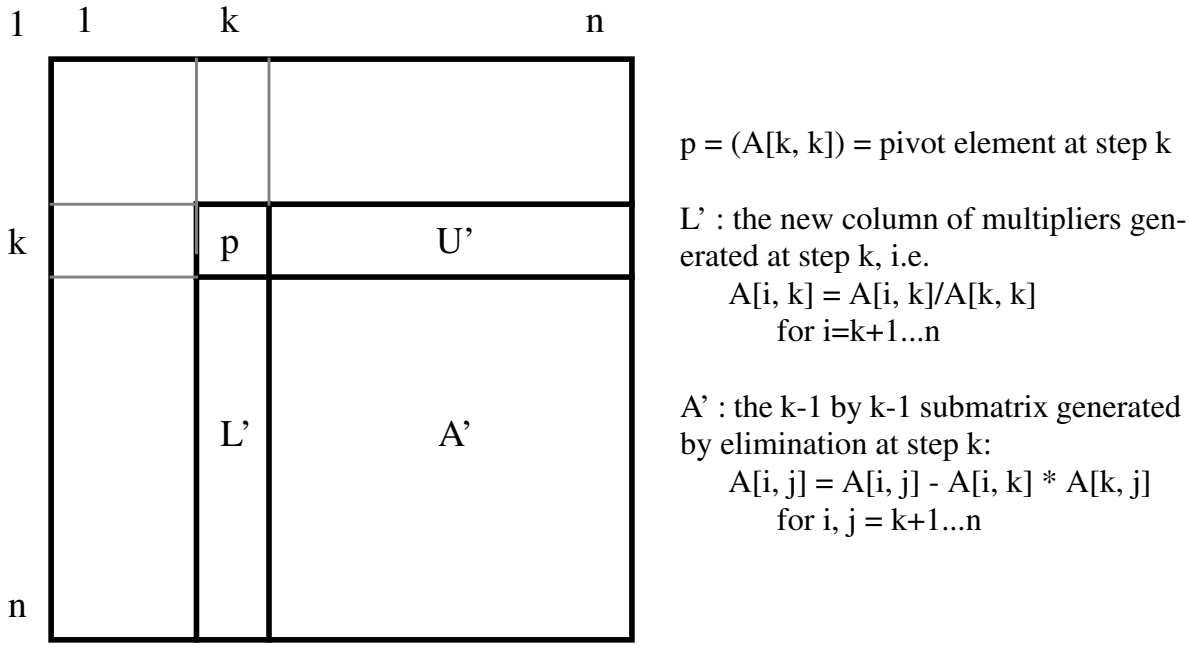


Figure 4: LU-Decomposition using Gaussian elimination

The algorithm proceeds in n steps. At each step k, it uses the pivot element p at $A[k,k]$ to generate a new partial column L', then using elements from the current partial row U' it performs elimination on the $(n-k)$-by-$(n-k)$ submatrix.

In programs written in a conventional language such as Fortran, the new data for L and U overwrite corresponding locations of A and are unchanged for the remainder of the computation. Using previous proposals of monolithic array constructors, coding Gaussian elimination may involve the creation of a sequence of n−1 intermediate arrays, which may lead to excess overhead due to computation-less copying and wasted memory space. The problem was pointed out by Nikhil [30] in a discussion of his draft solution (see Figure 5). To avoid such problems, both Hudak and Nikhil have given solutions to the LU-decomposition using a different algorithm (the Crout method) [18, 31].

9 Monotonic Definition of an Iteration Variable

It is quite natural (to us at least) to specify the highly recursive computation of LU decomposition using explicit evaluation steps, as required by the Gaussian elimination method. We observe that the use of unnecessary intermediate arrays is due to the single assignment style of functional programming – at each step a new (temporary) array is defined. However, one of

```
def LUD A_initial N =
          -- As[1] is the original matrix to be decomposed.
          -- As[k] (k in 2..N) is an (N-k+1) by (N-k+1) matrix
          -- obtained by elimination from A[k-1] and L[k-1].
    { As = {vector (1,N)
            | [1]   = A_initial
            | [k+1] = elim As[k] L[k] k     || k <- 1 to N-1};

          -- L[k] (k in 1..N-1) is the column of multipliers from step k.
      L  = {vector (1,N-1)
            | [k]   = multipliers As[k] k   || k <- 1 to N-1};

          -- U[i] (i in 1..N) is just a copy of the top row of As[i].
      U  = {vector (1,N)
            | [i]   = {vector (i,N)
                       | [j] = As[i][i,j]   || j <- i to N}
                                            || i <- 1 to N}
      def multipliers Ak k
         = {vector (k+1,N)
            | [i]   = Ak[i,k]/Ak[k,k]       || i <- k+1 to N};

      def elim Ak Lk k
         = {matrix (k+1,N),(k+1,N)
            | [i,j] = Ak[i,j]-Ak[k,j]*Lk[i] || i <- k+1 to N
                                            &  j <- k+1 to N};
      in L,U }
```

Figure 5: LU-Decomposition written in Id-Nouveau

the main reasons for only allowing single assignment in applicative languages is to eliminate side-effects in programs (and keep referential transparency). Now let us ask what side-effects may appear if we eliminate those intermediate arrays.

First, let us begin with a simple example of an iterative definition of a scalar in Val:

```
x := for  i := 1;
          t := t1
     do if i = N then t
        else iter
          i := i+1;
          t := f(t)      % assume that f is some unary function
        enditer endif
     endfor
```

In this loop expression the internal name `t` will be bound to a sequence of intermediate values during the tail-recursive computation. However, `t` is not visible outside the loop. A compiler should not allocate an array to hold these intermediate values of `t` because only the final value may be used outside the loop. In a dataflow interpreter [10], `t` is carried by successive tokens during each iteration through the loop. The sequence of "overwrites" to `t` may be viewed as a sequence of operations which *monotonically* defines the final result.

Now let us apply these observations to provide for the monotonic definition of arrays using the new array constructor. This may be done by adding an iteration facility. Using this facility, the LU-decomposition can then be written as:

```
AA := for i,j in [1,n]
      construct AT iter by k from 1..n
      initial AT := A repeat

        [(n-k,1),(k+1,k)]:              % First process current column
          AT[i,k]/AT[k,k]               % by computing the multipliers.
        !
        [(n-k,n-k),(k+1,k+1)]:          % Then process the current row:
          AT[i,j]-(NEW AT)[i,k]*AT[k,j] % eliminate in n-k by n-k region

      end for
```

The line `construct AT iter by k from 1..n` means that the body of the code is to be executed once for each value of k using a single copy of the array AT. Right-hand-side references to AT refer to values as defined by the previous iteration (or the initial value). A

reference to `NEW AT` refers to values defined in the current iteration. References to elements that are not yet defined and overwrites of already defined elements are programming errors.

During each step of `k`, the two regions are evaluated in sequence (as specified by the sequential composition operator "`!`"): the computation of the column multipliers is performed first, followed by elimination of the remaining $(n-k)$-by-$(n-k)$ region.

Evaluation of array elements within either of the two regions can be done in parallel by default. Values of elements of AT outside these two regions will not change, thus avoiding useless copying. In each iteration, the specified binding regions and the default regions form a partition of the index domain of the array constructor.

Something like the above code could be written in Sisal using the iterative **for initial** loop with incremental update of the array. The advantage of our proposal is that it avoids the use of incremental array update operations, making it easy for a compiler to determine that only a single array need be allocated for the result.

10 Future Work

One problem which has not been fully addressed in this paper is the detection that the subregions making up a partition are mutually exclusive and collectively exhaustive. Neither have we proven that all array references can be shown conclusively at compile time to lie within or outside a defined region. Although both are clearly programming errors, we are seeking a solution which will ensure that such errors should be manifested by the implementation in a deterministic fashion for debugging purpose. We are currently evaluating existing solutions proposed for Haskell or Id Nouveau through runtime detection and resolution mechanisms, as well as investigating our own schemes. We plan to report our solution in another publication.

We plan to continue experimenting with the new array constructor on a number of representative benchmark programs. Based on these experiments, we hope to improve the design and study its implementation on the McGill dataflow architecture model [11].

A Syntax of the Monolithic Array Constructor

mono-array-constructor ::= `for` *index-list* `in` *domain-spec* `construct` [*id*]
mono-body
`end for`

domain-spec ::= `[` *shape* `,` *lowbd* `]`

mono-body ::= *region-constructor* [`;` *mono-body*]
| *region-constructor* [`!` *mono-body*]

region-constructor ::= *basic-constructor*
| *sequence-constructor*
| *iter-constructor*

basic-constructor ::= *region-spec* : *expression*

sequence-constructor ::= `ordered by` *id* `from` *range*
region-constructor

iter-constructor ::= `iter by` *id* `from` *range*
`initial` *id* `:=` *expr* `repeat`
region-constructor

region-spec ::= *simple-region-spec*
| *composite-region-spec*
| `otherwise`

simple-region-spec ::= `[` *shape* `,` *lowbd* `]`
| *point*

shape ::= `(` *int-expr-list* `)`

lowbd ::= *point*

point ::= `(` *int-expr-list* `)`

range ::= *int-expr* `..` *int-expr*

index-list ::= *id* [`,` *index-list*]

int-expr-list ::= *int-expr* [`,` *int-expr-list*]

int-expr	:=	integer-valued expression evaluated prior to array constructor
composite-region-spec	::=	*simple-region-spec region-op simple-region-spec*
	|	*simple-region-spec region-op hyperplane-spec*
	|	*hyperplane-spec region-op simple-region-spec*
region-op	::=	`&` | `|` | `-`
hyperplane-spec	::=	a linear function of the index variables corresponding to the enclosing region (see examples in the text)

References

1 W.B. Ackerman. A structure processing facility for data flow computers. In *Proceedings of the 1978 International Conference on Parallel Processing*, Aug. 1978.

2 W.B. Ackerman and J.B. Dennis. *VAL — A Value-Oriented Algorithmic Language.* Technical Report 218, Laboratory for Computer Science, MIT, 1979.

3 S. Anderson and P. Hudak. Compilation of Haskell array comprehensions for scientific computing. In *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (SIGPLAN Notices, vol. 25, no. 6)*, pages 137–149, June 1990.

4 Arvind, R.S. Nikhil, and K.K. Pingali. I-Structures: Data structures for parallel computing. *ACM TOPLAS*, 11(4):598–632, Oct 1989.

5 U. Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, Boston, Ma., 1988.

6 U. Banerjee. Unimodular transformations of double loops. In *Proc. of the Third Workshop on Programming Languages and Compilers for Parallel Computing*, Irvine, Ca., Aug 1990. To be published by Pitman in their series Monographs in Parallel and Distributed Computing.

7 D.C. Cann. *Compilation Techniques for High Performance Applicative Computation.* PhD thesis, Colorado State University, 1989.

8 W.P. Crowley, C.P. Henderson, and T.E. Rudy. *The SIMPLE code.* Technical Report UCID-17715, Lawrence Livermore National Laboratory, Feb 1978.

9 J. Darlington, P. Henderson, and D.A. Turner. *Functional Programming and its Applications — An Advance Course.* Cambridge University Press, 1982.

10 J.B. Dennis. First version of a data-flow procedure language. In *Proceedings of the Colloque sur la Programmation*, pages 362–376, Springler-Verlag, 1974.

11 G.R. Gao. A flexible architecture model for hybrid dataflow and control-flow evaluation. In *Proc. of the International Workshop: Dataflow — A Status Report*, in conjunction with the ACM Annual Symposium on Computer Architecture, Israel, May 1989. To be published by Prentice-Hall.

12 K. Gopinath and J.L. Hennessy. *Copy Elimination with Abstract Interpretation.* CLaSSiC Project Manuscript CLaSSiC-87-17, Stanford University, 1987.

13 K.P. Gostelow and R.E. Thomas. A view of dataflow. In *AFIPS Conf. Proc., vol. 48*, pages 629–636, 1979.

14 P.N. Hilfinger and P. Colella. *FIDIL: A Language for Scientific Processing.* Technical Report UCRL 98057, Lawrence Livermore National Laboratory, Jan 1988.

15 P. Hudak. *ALFL Reference Manual and Programmer's Guide.* Technical Report YALEU/DCS/TR-322, Yale University, July 1984.

16 P. Hudak. Arrays, non-determinism, and parallelism: A functional perspective. In *Graph Reduction*, pages 312–327, Springer-Verlag, LNCS-279, 1987.

17 P. Hudak. Conception, evolution, and application of functional programming languages. *Computing Surveys*, 21(3), Sept. 1989.

18 P. Hudak and S. Anderson. *Haskell Solutions to the Language Session Problems at the 1988 Salishan High-Speed Computing Conference.* Technical Report YALEU/DCS/RR-627, Yale University, Jan 1988.

19 P. Hudak and A. Bloss. The aggregate update problem in functional systems. In *Proc. of the 12th ACM Symp. on Principles of Prog. Lang.*, pages 300–314, 1985.

20 P. Hudak and P. Wadler (editors). *Report on the Programming Language Haskell, A Non-strict Purely Functional Language (Version 1.0).* Technical Report YALEU/DCS/RR777, Yale University, Department of Computer Science, April 1990.

21 American National Standards Institute. Fortran 8X draft. *Fortran Forum*, 8(4), Dec 1989.

22 F. Irigoin and R. Triolet. Supernode partitioning. In *Proc. of the 15th ACM Symp. on Principles of Prog. Lang.*, pages 319–329, 1988.

23 Kenneth E. Iverson. *A Programming Language.* John Wiley and Sons, New York, 1962.

24 R.M. Keller. *FEL (Function Equaation Language) Programmer's Guide.* Technical Report AMPS Technical Report 7, University of Utah, April 1983.

25 M. Lam and M. Wolf. Maximizing parallelism via linear loop transformations. In *Proc. of the Third Workshop on Programming Languages and Compilers for Parallel Computing*, Irvine, Ca., Aug 1990. To be published by Pitman in their series Monographs in Parallel and Distributed Computing.

26 Z. Li, P.-C. Yew, and C.-Q. Zhu. An efficient data dependence analysis for parallelizing compilers. *IEEE Trans. on Parallel and Distributed Systems*, 1(1):26–34, Jan 1990.

27 J.R. McGraw, S. Skedzielewski, et al. *SISAL: Streams and Iteration in a Single Assignment Language — Language Reference Manual Version 1.2.* Technical Report M-146, Lawrence Livermore National Laboratory, 1985.

28 T. More. Notes on the diagrams, logic and operations of array theory. In Bjorke and Franksen, editors, *Structures and Operations in Engineering and Management Systems*, Tapir Pub., Trondheim, Norway, 1981.

29 L.M.R. Mullin. *A Mathematics of Arrays.* PhD thesis, Syracuse University, 1988.

30 R.S. Nikhil. Solution to the LU decomposition problem posed at IFIPS WG 2.8 meeting. May 1989. Unpublished communication.

31 R.S. Nikhil and Arvind. *Id: A Language with Implicit Parallelism.* Computation Structures Group Memo 305, Laboratory for Computer Science, MIT, 1990.

32 R.S. Nikhil, K. Pingali, and Arvind. *Id Nouveau.* Computation Structures Group Memo 265, Laboratory for Computer Science, MIT, 1986.

33 S.L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall International Series in Computer Science, 1987.

34 J.E Ranelletti. *Graph Transformation Algorithms for Array Memory Optimization in Applicative Languages.* PhD thesis, University of California, Davis, 1988.

35 G.W. Stewart. *Introduction to Matrix Computation.* Academic Press, Inc., 1973.

36 P.L. Wadler. A new array operations. In *Graph Reduction*, pages 328–335, Springer-Verlag, LNCS-279, 1987.

37 D. Wise. Matrix algebra and applicative programming. In G. Kahn, editor, *Funct. Prog. Lang. and Comp. Arch.*, pages 301–324, Springer-Verlag, LNCS-274, 1987.