Department of Computer Science

Uniqueness Analysis of Array Comprehensions Using the Omega Test

David Garza and Wim Bohm

Technical Report CS-93-127

October 21, 1993

Colorado State University

Uniqueness Analysis of Array Comprehensions Using the Omega Test [1]

David Garza and Wim Böhm

Department of Computer Science
Colorado State University
Fort Collins, CO 80523
tel: (303) 491-7595
fax: (303) 491-6639
email: *bohm@cs.colostate.edu*

Abstract

In this paper we introduce the uniqueness problem of array comprehensions. An array comprehension has the uniqueness property if it defines each array element at most once. Uniqueness is a necessary condition for the correctness of single assignment languages such as Haskell, Id, and Sisal. The uniqueness problem can be stated as a data dependence problem, which in itself can be reformulated as an integer linear programming problem. We derive algorithms to solve this problem using the Omega test, an Integer Linear Programming tool.

1 Introduction

One of the major applications of supercomputers is scientific numerical computing where much time is spent in performing array computations. This suggests that a language intended for scientific numerical computation must have a very efficient implementation of array operations as one of its key features. Functional languages provide an implicitly parallel, machine independent programming paradigm, avoiding many of the problems of explicit, machine dependent, and non-deterministic programming associated with explicitly parallel imperative languages. Some functional languages, such as Haskell, Id, and Sisal [4, 10, 13], have been designed to be used for scientific computing. Even though there have been significant improvements in the implementation of arrays for these languages [6, 2, 7, 1], there are still several problems that have not been addressed.

An *array comprehension* is a functional monolithic array constructor, defining an array as a whole entity. Id and Haskell have incorporated recursive array comprehensions, where array

[1] This work is supported in part by NSF Grant MIP-9113268

```
A = { 1D_array((1,n),(1,n)) of
      | [1,j] = 1                          || j <- 1 to n
      | [i,1] = 1                          || i <- 2 to n
      | [i,j] = A[i-1,j] + A[i,j-1]        || i <- 2 to n ; j <- 2 to n
    }
```

Figure 1: Array Comprehension for the Pascal Triangle in Id

elements can be defined in terms of other array elements of the same array. Moreover, Id and Haskell arrays are non-strict, i.e., not all elements of the array need to be defined. Sisal 2 [4] has incorporated the simpler form of non-recursive array generator. Sisal arrays are strict, i.e., they can be completely defined before any of the array elements needs to be used.

An example of a non-strict Id style array comprehension for the pascal triangle is given in figure 1. In the first line of figure 1 the dimensionality and bounds of the array are defined. A *region* of the form: *[target] = expression || generator* is equivalent to the loop construct *for generator do array[target] = expression.*

The semantics of array comprehensions obeys the *single-assignment* rule of functional languages, which prescribes that an array-element may not be defined more than once. In the current implementation of Id a redefinition of an array element will give rise to a run-time error [10]. Checking for this error introduces run-time inefficiency in most implementations of Id. We say that an array is *uniquely defined* if non of its elements is defined more than once. Compile time *uniqueness analysis* avoids the inefficiency of run-time checks. Uniqueness analysis is a form of array dependence analysis, and therefore employs subscript analysis techniques, similar to those used in optimizing and parallelizing conventional language compilers[15]. We have chosen the Omega test [11], which is based on integer linear programming, for this work. We will derive algorithms that turn an array comprehension into an integer linear programming problem that then will serve as input for the Omega test.

The rest of this paper is organized as follows. Section two gives a brief description of array comprehensions. Section three introduces the Omega test. Section four presents an algorithm for checking bounds. Section five presents algorithms for uniqueness analysis with some examples. Section six discusses related and future research. Section seven provides concluding remarks.

2 Array comprehensions

As mentioned in the introduction, array comprehensions are a form of monolithic array definitions, as opposed to constructs where an empty array is declared and its element values are defined throughout the program. Besides being an elegant way of defining an array, array comprehensions can have some efficiency advantages over other functional array constructors. Consider an array with the diagonal elements equal to 1, and the rest of the elements equal to the sum of row and column index. A possible way for defining this array is:

```
{ def fill (i,j) = if (i == j) then 1 else i+j
  In  make-matrix( (1,100), (1,100) fill}
```

This definition involves the run time evaluation of the conditional for every element of the matrix being created. The same matrix can be defined with the next array comprehension without incurring this run-time overhead.

```
{2D_array((1,100), (1,100)) of
 | [i,j] = i + j || i <- 1 to 99 ; j <- (i+1) to 100
 | [i,j] = i + j || i <- 2 to 100 ; j <- 1 to (i-1)
 | [i,i] = 1     || i <- 1 to 100 }
```

Using array comprehensions we can create recursive data structures very naturally. An example of this is the pascal triangle computation shown in figure 1. The abstract syntax for an array comprehension creating an n-dimensional array consisting of m regions is:

$$
\begin{array}{l}
nD_array((l_1,u_1),\ldots,(l_n,u_n)) \text{ of} \\
\quad | \ [f_1^1(I_1),\ldots,f_1^n(I_1)] = expr_1 \ \langle \ || \ gen_1^1 \ ;\ldots; \ gen_1^{d_1} \rangle \\
\quad | \ [f_2^1(I_2),\ldots,f_2^n(I_2)] = expr_2 \ \langle \ || \ gen_2^1 \ ;\ldots; \ gen_2^{d_2} \rangle \\
\quad \vdots \\
\quad | \ [f_m^1(I_m),\ldots,f_m^n(I_m)] = expr_m \ \langle \ || \ gen_m^1 \ ;\ldots; \ gen_m^{d_m} \rangle
\end{array}
$$

where $\langle .. \rangle$ indicates option, and each generator expression gen_j^k , is of the form: $i_j^k \leftarrow l_j^k \ to \ u_j^k$. Zero or more generator expressions define a region using a cross product of nested loops. I_j is the vector of loop variables for region j. The loop variables of region j are called i_j^k $(1 \leq k \leq d_j)$. The bounds of a loop variable may use previously defined loop variables. The expression f_j^p, $(1 \leq p \leq n)$ defines the subscript expression in the p-th dimension of region j. In the formulation above we have assumed that the step of the generator expressions is always 1. Techniques for obtaining a step 1 for each loop variable can be found in e.g. [15].

3 The Omega Test

A significant number of data dependence tests [3, 15, 5] assume a predefined "standard" order of computation [5]. In array comprehensions we do not have such predefined order. The Omega test [11, 12] is an exact data dependence test, free of assumptions on order of evaluation, and based on integer linear programming techniques. Although it has a worst-case exponential time complexity, this rarely occurs when using it for data dependence analysis. In fact, the time needed by the Omega test to analyze a problem is rarely more than twice the time required to scan the array subscripts and loop bounds. The Omega test can work with symbolic values, and it can also be used to simplify integer programming problems instead of just deciding them.

An integer linear programming problem consists of a number of equalities and inequalities of the form: $\sum_{i=1}^{n} a_i x_i = c$ and $\sum_{i=1}^{n} a_i x_i \geq c$, where a_i $(1 \leq i \leq n)$ and c are constants and x_i $(1 \leq i \leq n)$ are variables. Given an integer linear programming problem P, the Omega test decides whether there is an integer solution to P, and if so, the values of the variables that satisfy the constraints are produced. A brief description of the omega test follows.

First each constraint is normalized, such that the gcd (greatest common divisor) of all the a_i $(1 \leq i \leq n)$ coefficients of the constraint is equal to 1. At this stage the traditional gcd test [3] can be used to check whether no solution exists. Normalized constraints are eliminated in an iterative process by forcing a coefficient of 1 on some variable which can then be eliminated. After eliminating the equality constraints, we can check for contradictions in the remaining constraints.

We eliminate variables from our set of inequalities one at a time until we are able to prove or disprove a solution to the problem. The Omega test uses Fourier-Motzkin variable elimination, which finds the n-1 dimensional shadow cast by an n-dimensional object. If there are no integer points in this so called *real shadow*, then no integer solution exists. However, if there are solutions in the real shadow we cannot guarantee that the original problem has solutions.

If the solution was not disproved in the previous steps, an adaptation to Fourier-Motzkin computes the *dark shadow* which guarantees that for every integer point in the dark shadow there is an integer point in the object above it. However, if there is no solution in the dark shadow, there is still the possibility that the original problem has a solution. In this case the solution is closely nestled between an upper and lower bound, and a set of planes is generated such that a solution will lie on one of them. If after exhausting all possible lower bounds no solution is found, the problem has no solution.

4 Bounds Test

Before the uniqueness analysis we apply algorithm 1 which checks if a specific region of the array comprehension is defining elements out of the bounds of the whole array. This will simplify the uniqueness analysis and other forms of analysis which rely on the assumption that all the elements are defined within the bounds of the array.

Algorithm 1: Bounds Test

1. Set d to the number of loop variables in vector I of the region being tested, and n to the dimensionality of the array.

2. Generate vector $X = (x^1, x^2, x^3, \ldots, x^d)$. This vector represents the set of unknown loop variables for which we will try to find an integer solution.

3. For each element e^k $(1 \leq k \leq d)$ of X generate constraints expressing that e^k falls in the appropriate loop bounds. These constraints are of the form $l^k \leq e^k \leq u^k$ where l^k and u^k are the upper and lower bounds of the loop variable i^k.

   We will call P the integer programming problem resulting from the constraints defined in this step.

4. For p from 1 to n, create a problem L_p obtained by adding the following constraint to P:

   $$f^p(X) < l_p$$

   l_p is the lower bound of the p-th dimension of the array.

5. For p from 1 to n, create a problem U_p obtained by adding the the following constraint to P:

   $$f^p(X) > u_p$$

   u_p is the upper bound of the p-th dimension of the array.

The omega test is used to check if a solution exists to any of the L_p or U_p problems. If no solution is found then we say that the current region defines array elements within its bounds.

An example of use of this algorithm is given in the following section.

5 Uniqueness Analysis

The algorithms in this section describe how to transform an array comprehension into a linear integer programming problem representing a uniqueness problem. When checking for uniqueness, we search for output dependence between any two definitions in the array comprehension. We know that for any two n-dimensional array references $s_x : a(f_x^1(I_x), \ldots, f_x^n(I_x))$ and $s_y : a(f_y^1(I_y), \ldots, f_y^n(I_y))$, there is an output dependence between s_x and s_y if and only if $f_x^1(I_x) = f_y^1(I_y) \& , \ldots, \& f_x^n(I_x) = f_y^n(I_y)$.

There can be two forms of output dependence. Elements in one region can be defined more than once. This occurs when s_x and s_y are the same expression. The second source of dependence is when we have a dependence between any two definitions from different region definitions. This occurs when s_x and s_y are two different expressions. Therefore we can split uniqueness analysis into two subproblems:

- **Intra-regional uniqueness**: identifies whether there is a redefinition of an array element in the same region.
- **Inter-regional uniqueness**: identifies whether there is a redefinition of an array element between any two different regions.

We say that an array comprehension has the uniqueness property if and only if all its regions are intra-regional unique and the array is also inter-regional unique.

5.1 Intra-regional Uniqueness

Algorithm 2: Intra Regional Uniqueness Test

1. Set d to the number of loop variables in vector I of the region being tested, and n to the dimensionality of the array.
2. Generate two vectors $X = (x^1, x^2, x^3, \ldots, x^d)$ and $Y = (y^1, y^2, y^3, \ldots, y^d)$. These vectors represent the set of unknown loop variables for which we will try to find an integer solution.
3. For each element e^k $(1 \leq k \leq d)$ of X and Y generate constraints expressing that e^k falls in the appropriate loop bounds. These constraints are of the form $l^k \leq e^k \leq u^k$ where l^k and u^k are the upper and lower bounds of the loop variable i^k.
4. For each subscript expression f^p $(1 \leq p \leq n)$ generate the equality that represents the test for dependence:

$$f^p(X) = f^p(Y)$$

$f^p(X)$ and $f^p(Y)$ are obtained from $f^p(I)$ by variable replacement of each instance of i^k of the I vector by x^k or y^k. The integer programming problem resulting from the constraints defined in the previous steps is called P.

5. For k from 1 to d, create a problem P_k obtained by adding the constraint $x^k < y^k$ to P.

The Omega test is used to check if a solution exists to any of the P_k integer programming problems. If no solution is found, we declare the region intra-regional unique.

5.1.1 Example

Consider the following array comprehension

```
A = {2D_array ((1,75),(1,75) of
  | [2i+1,j]     || i <- 0 to 25 ; j = i+1 to 50                          %region 1
  | [2*k,2*k+j]  || i <- 1 to 4 ; k <- i+1 to 2i ; j <- 2*k+1 to i+2*k} %region 2
```

For region 1, the vector of loop variables is $I_1 = (i, j)$ with $0 \leq i \leq 25$ and $i + 1 \leq j \leq 50$ and index expressions $f_1^1(I_1) = 2i + 1$ and $f_2^1(I_1) = j$.

We first check bounds using algorithm 1. Step 2 of the algorithm will create the vector $X = (x_1, x_2)$. Step 3 creates the following constraints:

$$0 \leq x_1 \leq 25, \quad x_1 + 1 \leq x_2 \leq 50.$$

Step 4 will add the constraint $2x_1 + 1 < 1$ to P yielding problem L_1. It will also add the constraint $x_2 < 1$ to P yielding problem L_2. Step 5 adds the constraint $2x_1 + 1 > 75$ to P yielding problem U_1 and it also adds constraint $x_2 > 75$ yielding problem U_2. The omega test determines that there is no solution to any of the problems L_1, L_2, U_1, and U_2 therefore all the elements defined in region 1 are within the bounds of the array.

For region 2 the vector of loop variables is $I_2 = (i, k, j)$ with $1 \leq i \leq 4$, $i + 1 \leq k \leq 2i$ and $2k + 1 \leq j \leq i + 2k$ and the index expressions are $f_1^2(I_2) = 2k$ and $f_2^2(I_2) = 2k + j$. Step 2 of the algorithm will create the vector $X = (x_1, x_2, x_3)$. Step 3 creates the following constraints:

$$1 \leq x_1 \leq 4, \quad x_1 + 1 \leq x_2 \leq 2x_1, \quad 2x_2 + 1 \leq x_3 \leq x_1 + 2x_2$$

Step 4 will add the constraint $2x_2 < 1$ to P yielding problem L_1, It also adds the contraint $2x_2 + x_3 < 1$ to P yielding problem L_2. Step 5 adds the constraint $2x_2 > 75$ to P yielding problem U_1 and it also adds constraint $2x_2 + x_3 > 75$ yielding problem U_2. The omega test determines that there is no solution to any of the problems L_1, L_2, U_1, and U_2 therefore all the elements defined in region 2 are within the bounds of the array.

Now we proceed to check for uniqueness using algorithm 2. Step 2 of the algorithm will create the vectors $X = (x_1, x_2)$ and $Y = (y_1, y_2)$. Step 3 creates the following constraints:

$$0 \leq x_1 \leq 25, \quad x_1 + 1 \leq x_2 \leq 50, \quad 0 \leq y_1 \leq 25, \quad y_1 + 1 \leq y_2 \leq 50.$$

Step 4 adds $2x_1 + 1 = 2y_1 + 1$ and $x_2 = y_2$. All the above constraints define problem P. Step 5 adds the constraint $x_1 < y_1$ to P yielding problem P_1. The Omega test, determines that P_1 has no solution. We generate problem P_2 by adding the constraint $x_2 < y_2$ to P. The Omega test determines that there is no solution to problem P_2 either, and since we now have exhausted all the possible problems for this region, we can conclude that region 1 is intra-regional unique.

For region 2 step 2 creates vectors $X = (x_1, x_2, x_3)$ and $Y = (y_1, y_2, y_3)$. Step 3 creates the constraints

$$\begin{array}{l} 1 \leq x_1 \leq 4, \quad x_1 + 1 \leq x_2 \leq 2x_1, \quad 2x_2 + 1 \leq x_3 \leq x_1 + 2x_2 \\ 1 \leq y_1 \leq 4, \quad y_1 + 1 \leq y_2 \leq 2y_1, \quad 2y_2 + 1 \leq y_3 \leq y_1 + 2y_2 \end{array}$$

Step 4 adds $2x_2 = 2y_2$ and $2x_2 + x_3 = 2y_2 + y_3$.

All the above constraints define integer programming problem P.

Step 5 adds the constraint $x_1 < y_1$ to P resulting in problem P_1. The Omega test determines that there is a solution to this problem.

Region 2 defines array elements [8,17],[8,18],[10,21],[10,22],[10,23],[12,25], [12,26], and [12,27] more than once, and is therefore not intra-regional unique.

5.2 Inter-regional Uniqueness

Using the Omega test, algorithm 3 obtains the summary of array references for each of the region definitions of the array comprehension and then checks if there is an overlap between any of these array references.

Algorithm 3: Inter-regional Uniqueness Test

1. Set n to the dimensionality of the array and m to the number of regions in the array comprehension.
2. For each region r $(1 \leq r \leq m)$ perform steps (a) through (d)

(a) Set d to the number of loop variables in vector I_r.

(b) Generate inequality constraints based on the bounds of each element i_r^k $(1 \leq k \leq d_r)$ of vector I_r:

$$l_r^k \leq i_r^k \leq u_r^k$$

(c) Create n new variables x^p $(1 \leq p \leq n)$, and define the constraints on x_r^p in terms of the bounds of each of the dimensions of the original array:

$$l_p \leq x_r^p \leq u_p$$

(d) Create n equality constraints $(1 \leq p \leq n)$ to represent the relation between the index expression f_r^p $(1 \leq p \leq n)$ and the new variable x_r^p defined in step (c):

$$f_r^p(I_r) = x_r^p$$

This equality represents a summary of the array elements being accessed in region r.

3. For region r, steps (a), (b), (c), and (d) above define a problem P_r. For each combination of 2 regions, s and t, generate n equality constraints in terms of the variables x_s^p and x_t^p $(1 \leq p \leq n)$ created in step 2c.

$$x_s^p = x_t^p$$

P_{st} is the integer programming resulting from combining the constraints in P_s, P_t, and the constraints defined in this step.

If the Omega test finds that there is no solution to any of the P_{st} problems, we declare the array comprehension inter-regional unique.

Step 2b of the algorithm creates constraints that define the range of values that each loop variable can take on each of the different regions of the array comprehension. Step 2c defines new variables for each array dimension and defines the bounds for these variables. Step 2d finds a summary of the array references that are done on each array dimension for a particular region. Step 3 generates the constraints that check if there are any two overlapping regions.

5.2.1 Example

We apply algorithm 3 to the following array comprehension:

```
A={ 1D_array(1..2m) of  | [1] = 1                          %region 1
                        | [2*i] = i   || i = 1 to m        %region 2
                        | [2*j+1] = j || j = 1 to m-1}     %region 3
```

Step 2 generates the following constraints: for region 1 step 2b generates no constraints, step 2c creates the constraint $1 \leq x_1 \leq 2m$ and step 2d produces the constraint $1 = x_1$. Similarly, for region 2 the constraints are $1 \leq i \leq m$, $1 \leq x_2 \leq 2m$, and $2i = x_2$. For region 3 the constraints are $1 \leq j \leq m-1$, $1 \leq x_2 \leq 2m$ and $2j+1 = x_3$. Step 3 defines problem P_{12} by taking all the constraints generated for regions 1 and 2 and adding the constraint $x_1 = x_2$. This problem is given to the Omega test, which determines that there is no solution. We generate problem P_{13} in the same way. Again the Omega test finds no solution to this problem. The last problem generated is P_{23}, once again the Omega test determines that there is no solution. Since we have exhausted all possible combinations of two regions we conclude that the array is inter-regional unique.

Now we make a slight change to the array comprehension and set the index expression of the first region to 2. The Problem P_{12} willconsist of the following constraints:

$$1 \leq x_1 \leq 2m, \quad 2 = x_1, \quad 1 \leq i \leq m, \quad 1 \leq x_2 \leq 2m, \quad 2i = x_2, \quad x_1 = x_2$$

The Omega test finds a solution to this problem ($x_1 = x_2 = 2$), meaning that there is a redefinition of array elements.

5.3 Compiler Interface

Our algorithms require certain information that can be obtained from the program text. A simple data structure can be used to make available all the information required. This data structure should contain, among other information, the following fields:

- *Array_Id*: Array Identifier that uniquely identifies the array.
- *Dimension*: The dimensionality of the array.
- *Num_Regions*: The number of regions of the array comprehension.

- *Bounds*: A pointer to a data structure which contains, for each dimension of the array, the values of the upper and lower bounds.
- *Region_Info*: A pointer to a data structure that contains the following information specific to each region:
  - *Num_Vars*: The number of loop variables used in the region.
  - *Vars_Info*: A pointer to a data structure that contains *Num_Vars* tuples and each tuple consists of a variable identifier for the loop variable, and the upper and lower bounds of that variable.
  - *Subscript_Expr*: A pointer to a matrix similar to the *atom* data structure described in [9], where each row corresponds to one dimension of the array and each column corresponds to one of the *Num_Vars* loop variables of the region plus two extra columns: one that indicates if the subscript expression is linear and the other for the constant term. Each entry in row d column j, is the coefficient of the loop variable j for the subscript expression in dimension d.

Given this information our algorithm can extract the data needed and use the Omega test which has an interface that consists of several data structures, procedures and functions. The main data structure of the Omega test interface is one that defines the problem, some of the information contained here is the number of variables, number of equalities, number of inequalities, an array of equalities and greater than inequalities each represented by a data structure similar to the *Subscript_Expr* field above described.

6 Related and Future Research

When testing for inter-regional uniqueness we can think of each of the regions as a procedure call in an imperative language that defines certain elements of a globally defined array. Typical methods for testing data dependence in the presence of procedure calls base their analysis on obtaining a summary of the array references of each procedure and then testing for overlap between any of these array elements [14, 8, 5, 9]. One problem with these approaches is that except for [5] and [9] the approaches produce an approximate summary of the array references. For our problem we require precise information.

Burke and Cytron [5] propose to linearize the array space and to generate a list of array access information for each procedure. In order to prove independence between the array region accessed by procedure A and the array region accessed by procedure B, one needs to generate all possible pairs obtained by combining each of the elements of the list of array

accesses from procedure A with each element from the list of procedure B, and check the independence of all pairs.

Li and Yew [9] approach is very similar to Burke and Cytron's since they also form a set of array references and then apply a standard dependence test to prove independence between any two pairs of references. Two main differences are that they don't linearize the array space, mainly because data dependence tests are less precise when linearization has been applied. Secondly they introduce a data structure called *atom* which contains information about the array references and it is used to propagate this information to the calling procedure.

Hudak and Anderson [1] propose the use of subscript analysis for functional monolithic arrays. They recognize the uniqueness problem which they call Detecting Write Collisions, and they propose the use of Banerjee Inequalities test to check for independence. However, since this test is inexact they have to make pessimistic assumptions when the test is not able to disprove dependence.

Besides uniqueness analysis there are other compile time checks that can be performed to reduce run-time inefficiencies of functional arrays. Currently we are studying the following problems:

- **Completeness Analysis**: An efficiency problem of strict arrays, previously identified by Hudak and Anderson [1], is that a check is needed to ensure that the whole array is defined. Either run time checks, or static *Completeness analysis* are required in order to verify this.

- **Well-definedness**: For non-strict arrays an error will occur if the array comprehension tries to use array elements that never will be bound. This problem can be avoided by performing a well- definedness analysis.

- **Order of evaluation**: Some implementations of functional arrays rely on dynamic element level synchronization, like a per array element "presence-bit". Computations that use array elements will be synchronized by checking the presence-bit. This approach clearly causes run-time overhead, especially in machines without hardware support for presence bits. Also, for this approach to work, all processes defining an array element need to be started up at the same time, which causes high resource usage. If we are able to perform static order of execution analysis, we can schedule the array computations in such a way that the element level synchronization can be eliminated, and only the processes that can write an array element at some moment in the execution, will be started.

7 Conclusions

We have presented algorithms that check bounds and test for intra and inter regional uniqueness of array comprehensions for functional languages. Our algorithms use the Omega test as a tool. The Omega test was chosen because it is an exact, fast, and efficient and does not assume a standard order of evaluation. We have applied our algorithms to two array comprehension examples. The proposed algorithms should be applied to a more extensive number of examples in order to find possible practical limitations of the algorithm or of the Omega test itself. These limitations can lie in the exponential worst case complexity of the Omega test, or in the fact that bounds and index expressions must be linear.

Subscript analysis and program optimizations based on the information obtained from this type of analysis has been heavily used in imperative languages in order to improve parallelism and locality. We believe that functional languages can similarly benefit from subscript analysis, and this work is a first attempt that shows some of the benefits that can be obtained. We hope that more research in this direction can further help us to come up with optimizations and implementations of functional languages that will exploit parallelism and locality.

References

[1] Steven Anderson and Paul Hudak. Compilation of Haskell Array Comprehensions for Scientific Computing. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 137-149, June 1990.

[2] Arvind and Rishiyur S. Nikhil. I-Structures: Data Structures for Parallel Computing. *ACM Transactions on Programming Languages and Systems*, 11(4):598–632, October 1989.

[3] Utpal Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishing, 1988.

[4] A.P.W. Böhm, D. C. Cann, J. T. Feo and R. R. Oldehoeft. SISAL 2.0 Reference Manual. Technical Report CS-91-118, Computer Science Department, Colorado State University, Fort Collins, CO, November 1991.

[5] Michael Burke and Ron Cytron. Interprocedural Analysis and Parallelization. In *ACM SIGPLAN '86 Symposium on Compiler Construction*, pages 162-175, June 1986.

[6] D. C. Cann. *Compilation Techniques for High Performance Applicative Computation.* Ph.D. thesis, Colorado State University, Computer Science Department, Fort Collins, CO, 1989.

[7] G. R. Gao and Robert Kim Yates. An Efficient Monolithic Array Constructor. ACAPS Technical Memo 19, School of Computer Science, McGill University, Montreal, Canada, June 1990.

[8] Paul Havlak, Ken Kennedy. Experience with Interprocedural Analysis of Array Side Effects. In *Supercomputing '90*, pages 952-962, 1990.

[9] Zhiyuan Li and Pen-Chung Yew. Efficient Interprocedural Analysis for Program Parallelization and Restructuring. In *ACM SIGPLAN PPEALS*, pages 85-99, 1988.

[10] R.S. Nikhil, *Id (version 90.0)* Reference Manual. TR CSG Memo 284-1, MIT LCS 1990.

[11] William Pugh. The Omega Test: a fast and practical integer programming algorithm for dependence analysis. In *Supercomputing 1991*, pages 4-13, November 1991.

[12] William Pugh. A Practical Algorithm for Exact Array Dependence Analysis. *Communications of the ACM*, 35(8):102–114, August 1992.

[13] Boleslaw K. Szymanski. *Parallel Functional Languages and Compilers.* ACM Press, 1991.

[14] Rimi Triolet, Francois Irigoin, and Paul Feautrier. Direct Parallelization of Call Statements. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, pages 176-185, June 1986.

[15] Hans Zima with Barbara Chapman. *Supercompilers for Parallel and Vector Computers.* ACM Press, NY, 1990.