# Uniqueness and Completeness Analysis of Array Comprehensions

David Garza and Wim Bohm

Technical Report CS-94-113

May 6, 1994

# Uniqueness and Completeness Analysis of Array Comprehensions [1]

David Garza and Wim Böhm

Department of Computer Science
Colorado State University
Fort Collins, CO 80523
tel: (303) 491-7595
fax: (303) 491-2466
email: *bohm@cs.colostate.edu*

**Abstract**

In this paper we introduce the uniqueness and completeness problems 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 correctness in 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 uniqueness using the Omega Test, an Integer Linear Programming tool. An array comprehension has the completeness property if all its elements are defined. Completeness is a necessary condition for strict arrays. We present an algorithm that tests for completeness and describe an implementation of the algorithm based on multivariate polynomials.

**Keywords:** completeness, uniqueness, array dependence analysis, integer linear programming, Omega test, functional languages

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

## 1 Introduction

Some functional languages, such as Haskell, Id, and Sisal [12, 15, 14], have been designed to be used for scientific computing, and should therefore have efficiently implementable array operations [8, 3, 10, 2]. 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 elements can be defined in terms of other array elements of the same array. Id and Haskell arrays are non-strict, i.e., not all elements of the array need to be defined. Sisal 2 [6] has incorporated the simpler form of non-recursive array generator. Sisal arrays are strict, i.e., they must be completely defined.

```
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

An example of an Id style array comprehension for part of the "pascal triangle" is given in figure 1. In the first line, the dimensionality and bounds of the array are defined. The following lines define *regions* of the array. A region the form *[target] = expression* $\|$ *generators* is equivalent to the loop *for generators do array[target] = expression.* The syntax for an array comprehension creating an $n$-dimensional array consisting of $m$ regions is:

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

where $\langle .. \rangle$ indicates option. 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 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 expressions $l_j^k$ and $u_j^k$ of a generator 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 $j$.

Array comprehensions must obey the *single-assignment* rule of functional languages, which prescribes that an array-element may not be defined more than once. We say that an array is *uniquely defined* if none of its elements is defined more than once. In the current implementation of Id a redefinition of an array element will give rise to a run-time error [15]. Checking for this error introduces run-time inefficiency, which can be avoided by compile time *uniqueness analysis.*

Dynamically checking for the availability of array elements, for instance in the program in figure 1, has serious detrimental effects on performance. First of all, $n^2$ processes, one for each array element, are started up, even though only $O(n)$ parallelism can be exploited in this code. Secondly, because the processes are started up before data is available, most array element reads will occur before the corresponding write with the consequence that many read requests will have to be deferred (enqueued and dealt with later). Thirdly, individual reads and writes are more costly as they need to check presence bits. As already stated, a read request needs to be deferred when the element is not available. A write needs to generate an error message in the case the element was already written, and deal with deferred reads otherwise. Our analysis obviates the need for all this run time overhead, and will also avoid run time bounds checks.

Uniqueness analysis employs subscript analysis techniques, similar to those used in optimizing and parallelizing conventional language compilers[20]. A significant number of data dependence tests [4, 20, 7] assume a predefined "standard" order of computation [7]. In array comprehensions we do not have such predefined order. The Omega test [16, 17] is an exact data dependence test based on integer linear programming. It is free of assumptions on evaluation order. The Omega test can work with symbolic values, and it can also be used to simplify integer programming problems instead of just deciding them. Even though the Omega test has a exponential worst case complexity, it runs quite efficiently. For example, to determine dependence direction vectors for several programs from the NAS benchmark, the Omega test takes on the average less than 500 $\mu sec$ on a 12 MIPS Decstation 3100 [16]. In all cases that we have encountered, the Omega test takes not more than twice the time required to scan the array subscripts and loop bounds.

We will derive algorithms that turn a uniqueness problem of an array comprehension into an integer linear programming problem that then will serve as input for the Omega test.

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.

The uniqueness problem exists for Sisal style strict array generators as well as for Id style non-strict array comprehensions. An added problem of strict arrays 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. We present a compile time completeness test.

The rest of this paper is organized as follows. Section two presents an algorithm for checking bounds. Section three presents algorithms for uniqueness analysis with some examples. Section four presents the completeness test and discusses limitations of the approach. Section five presents a compiler interface required for our completeness and uniqueness analysis. Section six discusses related and future research. Section seven provides concluding remarks.

## 2 Bounds Test

Before the uniqueness test 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 and completeness tests 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 $e^k$ $(1 \le k \le d)$ of $X$ generate constraints expressing that $e^k$ falls in the appropriate loop bounds. These constraints are of the form $l^k \le e^k \le u^k$ 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 the $L_p$ or $U_p$ problems. If no solution is found, then the region defines array elements within the array bounds.

## 3 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 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 an output dependence between two definitions from two regions. This occurs when $s_x$ and $s_y$ are two different expressions. The **intra-regional uniqueness** test checks whether there is a redefinition of an array element in the same region. The **inter-regional uniqueness** test checks whether there is a redefinition of an array element between any two different regions. An array comprehension has the uniqueness property if and only if all its regions are intra-regional unique and the array is inter-regional unique.

### 3.1 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.

### 3.1.1 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:

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

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:

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

Step 4 will add the constraint $2x_2 < 1$ to $P$ yielding problem $L_1$, It also adds the constraint $2x_2 + x_3 < 1$ 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 intra-regional uniqueness using algorithm 2. Step 2 will create the vectors $X = (x_1, x_2)$ and $Y = (y_1, y_2)$. Step 3 creates the following constraints:

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

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

$$\begin{array}{lll} 1 \le x_1 \le 4, & x_1 + 1 \le x_2 \le 2x_1, & 2x_2 + 1 \le x_3 \le x_1 + 2x_2 \\ 1 \le y_1 \le 4, & y_1 + 1 \le y_2 \le 2y_1, & 2y_2 + 1 \le y_3 \le y_1 + 2y_2 \end{array}$$

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.

## 3.2 Inter-regional Uniqueness

Algorithm 3 obtains a summary of array references for each of the region definitions 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 $r$ $(1 \le r \le m)$ 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 $i_r^k$ $(1 \le k \le d_r)$ of vector $I_r$:
   $$l_r^k \le i_r^k \le u_r^k$$
   (c) Create $n$ new variables $x^p$ $(1 \le p \le n)$ , 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.

### 3.2.1 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 $P_{12}$ will consist of the following constraints:

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

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

## 4 Completeness Analysis

After the uniqueness and bounds tests have been satisfactorily performed, the completeness test reduces to checking whether the size of the total array is equal to the sum of the sizes of all its regions, where the size of a region is defined as the size of its iteration space. For example, in figure 1 the size of array A is $n^2$, and the sizes of the three regions are $n$, $n-1$, and $(n-1)^2$, respectively. When we compare the size of the array against the sum of the sizes of the regions, we obtain that both expressions are equal to the polynomial $n^2$. Note that verifying this involves manipulating non-linear expressions. Also note, that we do not need to solve a non-linear equation, we merely need to check equivalence of two polynomials.

The size of the array is given by $\prod_{i=1}^{n}(u_i - l_i + 1)$, where $n$ is the dimensionality of the array. This product expands into a multivariate polynomial that consists of the sum of $3^n$ terms such that each term is of the from $(a_1 a_2 \cdots a_n)$ where $a_i$ is either $u_i$, or $-l_i$, or 1.

For a given region $j$ with $k$ loop variables, the upper and lower bounds $u_j^l$ and $l_j^l$ of the loop variables $i_j^l, (1 \le l \le k)$ are linear functions of previously defined loop variables. The iteration space of region $j$ consists of all the integer points defined by the following equations (leaving out subscript $j$ of loop variable $i$ for simplicity):

$$p_{1,1} \le i^1 \le q_{1,1}$$
$$p_{2,1} i^1 + p_{2,2} \le i^2 \le q_{2,1} i^1 + q_{2,2}$$
$$\vdots$$
$$p_{k,1} i^1 + \cdots + p_{k,k-1} i^{k-1} + p_{k,k} \le i^k \le q_{k,1} i^1 + \cdots + q_{k,k-1} i^{k-1} + q_{k,k}$$

In the above equations, $p_{x,y}$ and $q_{x,y}$ $(1 \le x, y \le k)$ are integer constant values, and $p_{x,x}$ and $q_{x,x}$ can be symbolic integer variables. Therefore the size of a region is computed by summing multivariate polynomials whose upper and lower bounds are multivariate polynomials:

$$\sum_{i^1=p_{1,1}}^{q_{1,1}} \cdots \sum_{i^{k-1}=p_{k-1,1}i^1+\cdots+p_{k-1,k-2}i^{k-2}+p_{k-1,k-1}}^{q_{k-1,1}i^1+\cdots+q_{k-1,k-2}i^{k-2}+q_{k-1,k-1}} (q_{k,1} i^1 + \cdots + q_{k,k-1} i^{k-1} + q_{k,k} - (p_{k,1} i^1 + \cdots + p_{k,k-1} i^{k-1} + p_{k,k}) + 1)$$

The first task of the completeness test is to derive symbolic expressions for the array and region sizes. These sizes are expressed as multivariate polynomials in the upper and lower bounds of the array declaration for the size of the whole array, and as multivariate polynomials of the upper and lower bounds of the generators for the sizes of the regions. The second task is to verify that for all possible lower and upper bounds of the array, the sum of the sizes of all the regions equals the size of the whole array.

The following algorithm tests for completeness of an array comprehension, it assumes that the uniqueness and bounds test have been satisfactorily applied to the array. We will call the array being tested $A$.

**Algorithm 4: Completeness Test**

1. Set $n$ to the dimensionality of the array, and $m$ to the number of regions.
2. Derive $\mid A \mid$, the size of the array space:

$$\mid A \mid = \prod_{i=1}^{n} (u_i - l_i + 1)$$

3. For each region $j$ $(1 \leq j \leq m)$ derive $\mid A(I_j) \mid$, the size of its iteration space as follows:
   (a) if vector $I_j$ is empty then

   $$\mid A(I_j) \mid = 1$$

   else

   (b) Set $k$ to the number of loop variables in vector $I_j$

   $$\mid A(I_j) \mid = \sum_{i_j^1 = l_j^1}^{u_j^1} \sum_{i_j^2 = l_j^2}^{u_j^2} \cdots \sum_{i_j^{k-1} = l_j^{k-1}}^{u_j^{k-1}} (u_j^k - l_j^k + 1)$$

   where $u_j^k$, and $l_j^k$ are linear expressions in the loop variables $i_j^r$ and the bounds $u_i$ and $l_i$ of $A$, $(1 \leq i \leq n, r < k)$.
4. Derive $\mid A(I) \mid$ as the sum of the sizes of all the regions of $A$:

$$\mid A(I) \mid = \sum_{j=1}^{m} \mid A(I_j) \mid$$

5. If $\mid A(I) \mid - \mid A \mid \equiv 0$ then the array comprehension is complete. Otherwise it is incomplete.

## 4.1 Implementation Details of the Completeness Algorithm

We have written a program that implements algorithm 4. This program takes as input the bounds of each dimension of the array and the upper and lower bounds of the loop variables of each region. As output we get the array space size, the size of each region, and the sum of the sizes of all the regions of the array. We also get a polynomial that represents the difference between the array space and the sum of the sizes of the regions.

When implementing step 3 of the algorithm, we can try to simplify this by creating the expanded form of the expression for the size of the iteration space for a certain number of nesting levels, for instance for levels 1 to 10. However, using Maple [9] as a tool to compute the expansion of the sum in symbolic form, we find that for the case of one loop the resulting expression consists of 3 terms, for a doubly nested loop the expression consists of 17 terms, for triply nested loops the expression consists 179 terms, and the expression for 4-deep nested loops has a total of 3059 terms. These numbers make the idea of hardcoding the expression impractical. However, the large number of terms occurs because each loop has its lower and upper bounds defined in terms of linear expressions of all its enclosing loop indices. Also, since the expansion of the sums is done in symbolic form, the actual simplification of terms that we can get in intermediate steps is very limited. In real programs the loop indices are not as complex as the general case, and hence the original sum to be expanded has a much simpler form, known at compile time allowing many intermediate simplifications. Therefore we decide to perform the expansion of the sum that computes the size of the iteration space as part of the implementation of the algorithm using the actual values that define the loop bounds.

The program implements primitive operations of multivariate polynomials needed to expand the sums and products used in algorithm 4. These operations are exponentiation, multiplication, addition, subtraction, simplification, and replacement of one variable of a polynomial by another variable or polynomial.

The program works as follows: It computes $\mid A \mid$ by doing simple additions and multiplications of polynomials. The value of $\mid A(I_j) \mid$ is then computed for each of the regions of the array. This computation is accomplished by first creating the polynomial over which the multiple sums should be applied using simple addition of polynomials. Then we repeatedly expand a sum, starting from the inner sum and proceeding outwards, exploiting the scope rules of the nested loop, which state that the bounds of an inner loop can depend on outer loop variables, but that the bounds of an outer loop cannot depend on inner loop variables. To expand a sum of a polynomial it is first normalized, that is, the sum is modified such that its lower bound is set to 1, then each of the terms of the polynomial is replaced by the expanded form of the sum of that term.

Once the sizes of all regions have been computed, we compute $\mid A(I) \mid$ by performing an addition of all the $\mid A(I_j) \mid$ polynomials. Finally a subtraction of $\mid A(I) \mid$ and $\mid A \mid$ is performed. If the result of this is the 0 polynomial then the array and iteration space are of the same size.

## 4.2 Examples

We apply algorithm 4 to the following examples which are unique and have all elements defined within the bounds of the array.

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

Step 2 gets the size of the array space $\mid A \mid = 2n+1$. Step 3 obtains the sizes for each region: $\mid A(I_1) \mid = 1, \mid A(I_2) \mid = n, \mid A(I_3) \mid = 1$, and $\mid A(I_4) \mid = n-1$. Step 4 computes $\mid A(I) \mid = 2n+1$. Finally step 5 computes $\mid A(I) \mid - \mid A \mid = 0$. Therefore we conclude that the array comprehension is complete.

If we modify the upper bound of the array dimension in the previous example and set it to $2*n+2$, then $\mid A \mid = 2n+2$ and in step 5 we get $\mid A(I) \mid - \mid A \mid = -1$, so we conclude that the array comprehension is incomplete.

In a more complex example, we apply the completeness test to the next array comprehension that defines 4 tetrahedral regions of a cube as shown in figure 2.

```
A = {3D_array((1,n),(1,n),(1,n)) of
  | [i,j,k] = 1     || i <- 1 to n ; j <- i+1 to n ; k <- 1 to j-i        % region 1
  | [i,j,k] = 2     || i <- 1 to n ; j <- 1 to i-1 ; k <- 1 to i-j        % region 2
  | [i,j,k] = 3     || i <- 1 to n ; j <- n-i+2 to n ; k <- 2n+2-i-j to n % region 3
  | [i,j,k] = 4     || i <- 1 to n ; j <- 1 to n-i ; k <- i+j to n }      % region 4
```

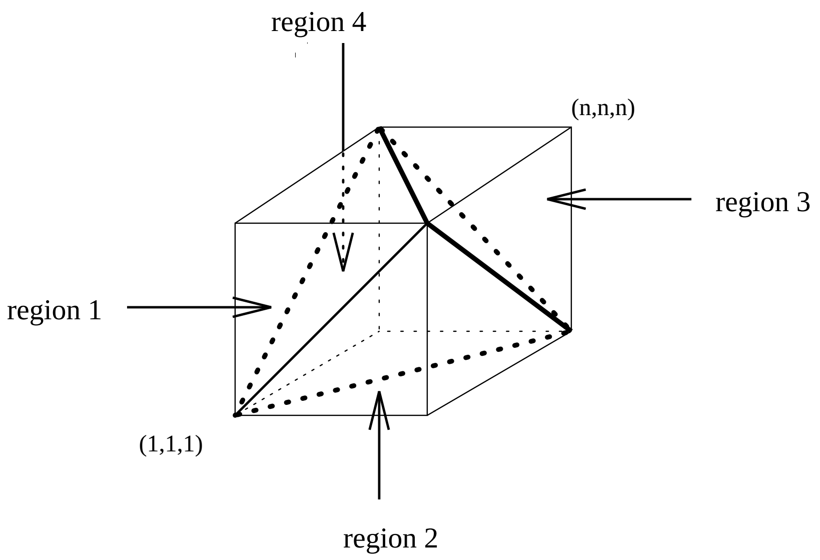

Figure 2: Cube with tetrahedral regions

Step 2 gets the size of the array space $\mid A \mid = n^3$. Step 3 obtains the sizes of each region. For example, the size of region 1 $\mid A(I_1) \mid$ is described by

$$\sum_{i=1}^{n} \sum_{j=i+1}^{n} (j - i - 1) + 1$$

which reduces to

$$\sum_{i=1}^{n} n^2/2 + n/2 - in - i/2 + i^2/2$$

which is further reduced to $n^3/6 - n/6$. Similarly, $\mid A(I_2) \mid = \mid A(I_3) \mid = \mid A(I_4) \mid = n^3/6 - n/6$. Step 4 computes $\mid A(I) \mid = 2n^3/3 - 2n/3$. Finally step 5 computes $\mid A(I) \mid - \mid A \mid = -n^3/3 - 2n/3$. Therefore we conclude that the array comprehension is incomplete. This is because the points in the inner tetrahedron delimited by the diagonals (the thick and dotted lines in figure 2) are not defined.

### 4.3 Limitations

An obvious limitation of both uniqueness and completeness tests is that they rely on the bounds and index expressions to be linear. Another limitation of algorithm 4 can be illustrated with the following example:

```
A = {1D_array((1,m)) of
  | [1]   = 2x+y
  | [i]   = i     || i <- 2 to n - 1
  | [n]   = z }
```

Here our algorithm obtains $\mid A \mid = m$ and $\mid A(I) \mid = n$ and since $\mid A(I) \mid - \mid A \mid = m - n$, we conclude that the array is incomplete. However, if $m = n$ then the array would be complete. Currently, when analyzing the array comprehension in vacuo, we don't have enough information about $m$ or $n$ to conclude completeness. In our compiler, constant propagation and induction variable replacement optimizations [1] must be performed prior to our completeness analysis, allowing us to have as much information as possible regarding symbolic variables used in the array comprehension.

When expanding the sums, the size of the integer coefficients in the multivariate polynomials grows very large very quickly. We therefore need to extend our implementation to support multiple precision integers.

## 5 Compiler Interface

Our algorithms require certain compile time information gathered in a record with 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 [13], 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. There are 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 and our completeness analysis tool. The omega test has an interface that consists of several data structures, procedures and functions. The main data structure defining the integer linear programming problem contains the number of variables, number of equalities, number of inequalities, and an array of equalities and 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 [19, 11, 7, 13]. One problem with these approaches is that except for [7] and [13] the approaches produce an approximate summary of the array references. For our problem we require precise information.

Burke and Cytron [7] 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's [13] approach is 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 is used to propagate this information to the calling procedure.

Hudak and Anderson [2] 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. They also identify the completeness problem which they called Detecting Empties. However, no detailed algorithm or tool to check for this is given.

Besides uniqueness and completeness 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:

- **Well-definedness**: For non-strict arrays an error will occur if the array comprehension tries to use array elements that never will be bound, e.g., because of circular dependencies. Well-definedness analysis can check for this.
- **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. This analysis will transform the programmer-defined regions of the array into a set of regions, which are intra regionally data independent. As an example, in figure 1, the user-defined regions are a row, a partial column and a submatrix, whereas the regions with intra regional data independence are anti-diagonals. We will determine the use of the Omega test to solve this restructuring problem.

## 7 Conclusions

We have presented algorithms that check bounds and test for intra and inter regional uniqueness and completeness of array comprehensions for functional languages. Our bounds and uniqueness 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. For our completeness analysis we have written a program that implements the completeness test by manipulating multivariate polynomials.

We have applied our uniqueness, bounds and completeness algorithms to some 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 algorithms 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. Also our current implementation of the completeness test should be extended to handle multiple precision integers.

Subscript analysis and program optimizations based on the information obtained from this type of analysis have 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 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] A.V. Aho, R. Sethi, and J. D. Ullman. *Compilers. Principles, Techniques and Tools.* Addison-Wesley, Reading MA, 1986.

[2] 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.

[3] 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.

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

[5] Utpal Banerjee. *Loop Transformations for Restructuring Compilers. The Foundations.* Kluwer Academic Publishing, 1993.

[6] 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.

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

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

[9] B.W. Char, K.O. Geddes, G.H. Gonnet, B.L. Leong, M.B. Monagan, S.M.Watt, *Maple V Library Reference Manual.* Springer- Verlag and Waterloo Maple Publishing, 1991.

[10] 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.

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

[12] P. Hudak et. al. Report on the programming Language Haskell - A non-strict, Purely Functional Language - version 1.0, Technical report, Yale University, April 1990.

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

[14] J. R. McGraw, S. K. Skedzielewski, S. J. Allan, R. R. Oldehoeft, J. Glauert, C. Kirkham, W. Noyce, and R. Thomas. SISAL: Streams and iteration in a single assignment language: Reference manual version 1.2., Manual M-146, Rev. 1, Lawrence Livermore National Laboratory, Livermore, CA, March 1985.

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

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

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

[18] Alexander Schrijver. *Theory of Linear and Integer Programming.* John Wiley & Sons, New York, New York, 1987.

[19] 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.

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