Efficient Compilation of HASKELL Array Comprehensions

Steven Anderson and Paul Hudak

Research Report YALEU/DCS/RR-693
March 1989
This work is supported in part by
DARPA grant N00014-88-K-0573.

Efficient Compilation of HASKELL Array Comprehensions

Steven Anderson
Paul Hudak

Department of Computer Science
Yale University
Box 2158 Yale Station
New Haven, CT 06520

March 10, 1989

Abstract

Monolithic approaches to functional language arrays, such as HASKELL *array comprehensions*, have several advantages over imperative language arrays and incremental functional arrays: evaluation order of elements is imposed only by data dependence, and there is no need for sophisticated analyses to detect situations where destructive update is safe. However, naive implementations of monolithic arrays are very inefficient. Even in a strict context, a recursively defined array, which arises commonly in scientific computing, must represent its elements as thunks if the compiler does not know a safe evaluation order.

Interestingly, imperative language arrays face a similar problem when they are compiled for vector and parallel machines. Imperative language arrays *overspecify* the evaluation order of array elements, and an analysis is needed to determine the true dependences in order to allow safe *parallel* evaluation. On the other hand monolithic functional arrays *underspecify* evaluation order, and an analysis is needed to determine the true dependences in order to allow safe *sequential* evaluation. Not knowing the true dependences among array elements, both analyses must make the pessimistic assumption of overestimating dependences, and both must resort to pessimistic compilation strategies to preserve semantics. Subscript analysis, originally developed for imperative language vectorizing and parallelizing compilers, can be adapted to efficient compilation of monolithic functional language arrays.

We first introduce array comprehensions and a new update function called `bigupd`, which combines some of the advantages of both the incremental and monolithic approaches. We then describe the sources of inefficiency in compiling functional arrays, and some ways of eliminating those inefficiencies. We list the kinds of inter-element dependences that can occur in functional arrays, and how knowledge of these dependences gained from subscript analysis permits us to compile efficient code. The main body of the paper presents the graph analysis of array dependences for loop scheduling. The final section presents the number-theoretic basis of subscript analysis, adopting assumptions appropriate to functional arrays.

1 Introduction

Most of the proposals for incorporating contiguous arrays into functional languages follow either an *incremental* or a *monolithic* approach [13]. Monolithic approaches such as *array comprehensions* in HASKELL [8] have several advantages over incremental approaches: evaluation order of elements is imposed only by data dependence, and there is no need for sophisticated analyses such as [7] and [5] to detect situations where destructive update is safe. Even so, a naive implementation of monolithic arrays still will not approach the efficiency of imperative language arrays. This paper proposes ways to eliminate most of these inefficiencies; in particular, how to eliminate the need for thunks in sequential compilation of monolithic arrays.

Let us say a monolithically defined array m is *in a strict context* if it is used in such a way that all of its elements are evaluated; thus any element being $\perp$ is equivalent to the entire array being $\perp$. Not knowing whether m is in a strict context requires an implementation in which the evaluation of array elements is "delayed," in which case we say that the elements are represented as "thunks." This could possibly be avoided if we were to perform a suitable strictness analysis over the program — such an analysis would be similar to non-flat strictness analysis of lists, but would be more sophisticated because of the desire to incorporate subscript analysis.

Determining or declaring that m is in a strict context is a good assumption for scientific computations, the most common use of arrays, and results in a simpler compilation strategy, *if m is not recursively defined.* In this situation the order in which m's elements are evaluated is irrelevant, and we can compile code that stores m's final element values directly, instead of representing them as thunks.

But even if array m in in a strict context, it may be *recursively defined*, so that some elements of m depend on other elements of m, thus requiring that we still represent the element values as thunks. Such arrays are very common in scientific computation; the LU and QR matrix factorizations are only two of a vast number of examples. To avoid this overhead the compiler must determine a sequential evaluation order that guarantees that computing one element will only require the results of other elements already computed. Imperative language arrays are efficient because they require the *programmer* to determine this evaluation order. We would like to retain the advantage of not requiring the programmer to specify the order, while improving on the efficiency of implementing such arrays.

Interestingly, imperative language arrays face a similar problem when they are compiled for vector and parallel machines. Imperative language arrays (and incremental functional arrays) *overspecify* the evaluation order of array elements, and an analysis is needed to determine the true dependences in order to allow safe *parallel* evaluation. On the other hand monolithic functional arrays *underspecify* evaluation order, and an analysis is needed to determine the true dependences in order to allow safe *sequential* evaluation.

Not knowing the true dependences among array elements, both analyses must make the pessimistic assumption of overestimating dependences: any data dependence consistent with the assumed evaluation order may be present. In the absence of better information, both must resort to pessimistic compilation strategies to preserve semantics: imperative languages must prohibit vectorization and parallelization; functional languages must employ expensive run-time representations such as thunks.

The imperative language vectorizing and parallelizing compiler community has done extensive work on subscript analysis to provide compilers with better knowledge about

dependences ([1,3,4,6,9,14]). Subscript analysis is not completely general, but it does work on a vast number of important scientific applications. This paper shows that subscript analysis can be adapted to compile highly efficient sequential implementations of monolithic functional arrays. It has not escaped our attention that these techniques can also be used in a vectorizing and parallelizing compiler for functional arrays.

We first introduce array comprehensions and a new block-style update function called **bigupd** , which combines some of the advantages of both the incremental and monolithic approaches. We then describe the sources of inefficiency in compiling functional arrays, and briefly indicate some ways of eliminating those inefficiencies. We list the kinds of inter-element dependences that can occur in functional arrays, and how knowledge of these dependences gained from subscript analysis permits us to compile efficient code. The main body of the paper presents the graph analysis of array dependences for loop scheduling, treating the single loop case in detail, and sketching the multi-loop case. The final section presents the number-theoretic basis of subscript analysis, adopting assumptions appropriate to functional arrays.

2 HASKELL array comprehensions

A HASKELL array is created by the constructor `Array` ([13], [8]):

```
a = Array (1,n) [ (i, f i) || i <- [1..n] ]

b = Array ((1,1), (m,n))
    [ ((i,j), g i j) || i <- [1..m], j <- [1..n] ]
```

The first argument is a pair of bounds; for a d-dimensional array, each bound is a d-tuple. The second argument is a list of array elements, which are pairs of the form $(index, value)$. Nondistinct element indices in the list cause a collision error. Typically, this list is expressed as a list comprehension. (Present a better description of list comprehensions.)

Wadler [11] presents a series of program transformations for list comprehensions from an obvious but inefficient definition to a more complex but efficient implementation that uses no intermediate lists. When an array comprehension is written using generators of the form `i <- [low,inc..high]` , Wadler's transformation can be easily adapted so that each generator is compiled as a DO loop, although in general the DOs can only put thunks in the elements. See [2] for details; if space permits, we will sketch the transformation in the final paper.

Array comprehensions that append several $(index, value)$ lists, such as `Array (...)` $(L_1$ `++` L_2 `++` $L_3)$, can be compiled as a sequence of non-nested DO loops. Notice that the semantics of the array constructor does not depend on the order of pairs in the list argument. We can treat the lists as unordered multisets and append as multiset union: `i <- [low,inc..high]` can be replaced by `i <- [high,-inc..low]` , and $(L_1$ `++` $L_2)$ can be replaced by $(L_2$ `++` $L_1)$.

If an array in a strict context is not recursively defined, the DO loops can directly evaluate and store the final element values. Even so, there are inefficiencies that do not arise in imperative language arrays, such as checking for collisions and empties (to be discussed in in the next section). But if an array in a strict context is recursively defined,

and we do not know a safe evaluation order, we must compile the DO loops to store thunks for each element, then go through the entire array forcing the thunks.

In this paper we adopt the syntactic sugaring `:=` to separate the components of an (*index*, *value*) pair. This improves legibility in complicated expressions, and stresses the resemblance of an array to a set of *let* definitions:

```
Array (1,n) [ i := f i || i <- [1..n] ]
```

3 Suggested extensions to array comprehensions

3.1 Forall

We introduce an alternative notation, `forall` , that is more flexible than list comprehensions. Like list comprehensions, this new notation contains one or more generators and predicates, lexically scoped from left to right. It also contains a list; the generator variables are treated as free variables over this list.

```
a = Array (1,n) forall i <- [1..n]; [ i := f i ]

b = Array ((1,1), (m,n))
    forall i <- [1..m], j <- [1..n]; [ (i,j) := g i j ]
```

The semantics of `(forall i <- L1; L2)` : L_1 is type (*list* α); map the function $\lambda i.L_2$ over the list L_1 to get a list of type (*list* (*list* α)); concatenate these lists to get the result, a list of type (*list* α). (Present a formal definition).

An array constructor taking a `forall` (*index*, *value*) list with `i <- [low,inc..high]` generators can be compiled into DO loops; the transformation from the simple semantics to the efficient implementation is nearly the same as the transformation for list comprehensions. Again we can view `forall` lists, in the context of an array constructor, as unordered multisets, and view list append as multiset union.

Why a new list notation? Our experience has been that list comprehensions, compared with `forall` , are not as expressive as we would like in two ways.

(1) List comprehensions make lexical scoping of generators easy, but they make other kinds of lexical scoping painful, such as using a *let* to express sharing a computation across inner loop iterations.

```
[ let k = g i; in E || i <- [1..n], j <- [1..i-1] ]
```

The subexpression (g i) has been abstracted out of the expression E, where perhaps it occurs multiple times. (g i) depends on i but not on j, but we either must rely on the compiler to hoist the code out of the j loop, or resort to something ghastly like this:

```
[ E || i <- [1..n], k <- [(g i)], j <- [1..i-1] ]
```

We find `forall` more expressive. The scoping of generators is more explicit. Generator scoping becomes consistent with *let* scoping rules, permitting us to use both naturally in the same expression. `forall` makes explicit the important idea (buried and implicit in list comprehensions) that every generator operates over list-valued (multiset-valued) function $\lambda i.L$, appending together list instances (unioning multiset instances):

```
forall i <- [1..n];
    let k = g i
    in forall j <- [1..i-1]; [ E ]
```

(2) List comprehensions make it impossible to use more than kind of *clause* (i.e., more than one kind of (*index*, *value*) expression) at the heart of a set of nested loops. For example, suppose we want one clause for the top row, another clause for the leftmost column, and a third clause for inside of an $m * n$ matrix?

```
a = Array ((1,1), (m,n))
        [ (1,j) := 1  ||  j <- [1..n] ]  ++
        [ (i,1) := 1  ||  i <- [1..m] ]  ++
        [ (i,j) := a!(i-1,j) + a!(i,j-1)
                   ||  i <- [1..m], j <- [1..n] ]
```

As a programmer, we may find it convenient to think of this array row-wise or column-wise, but there is no way to express these notions! Also, a programmer frequently uses his knowledge of how language constructs are implemented to achieve the same semantics with greater efficiency. Consider this version of the program:

```
a = Array ((1,1), (n,n))
        (forall j  <-  [1..m], [ (1,j) := 1 ] ) ++
        (forall i  <-  [1..m];
            [ (i,1) := 1 ] ++
            (forall j <- [1..n];
                  [ (i,j) := a!(i-1,j) + a!(i,j-1) ] ))
```

A programmer may realize that even for a relatively dumb compiler, this version will reduce loop overhead by sharing the i loop, and even more importantly, will give a much better cache and page hit ratio for row-major array storage.

In the first version the compiler must detect the opportunity for loop jamming the two i loops, and realize that this is a beneficial transformation. It is important that compilers be able to do such things, but it is frustrating that list comprehensions do not permit us to express the result of such a simple transformation directly. This rest of this paper suggests many transformations to array comprehensions that are impossible to express in list comprehensions, but are easy to express with `forall` .

3.2 Bigupd

Single-element (upd *m index value*) takes as arguments array m, and a single (*index*, *value*) pair. Multi-element `bigupd` m L generalizes by taking a list L of (*index*, *value*) pairs. Nondistinct indices in L cause a collision error. Treat L as an unordered multiset. L coded as list comprehension or `forall` with appropriate generators will be compiled as DO loops.

We have found that some problems, such as LU factorization with partial pivoting to preserve numerical stability, are nearly impossible to program with only a constructor such as *Array* that does monolithic creation. This problem is extremely tedious to code without some kind of incremental update, since we constantly need to specify that these large parts

of new array m_2 are the same as old array m_1, while these small parts of m_2 are actually new. The compiler also cannot see that in-place update of m_1 is possible without a sweeping interprocedural analysis.

However, the updates are more naturally coded as row-, column-, or submatrix-at-a-time. `bigupd` is incremental, but it also resembles a monolithic update. The operational advantages of `bigupd` are discussed in the next section.

4 Sources of inefficiency in array comprehensions

Honoring the ancient maxim that a program spends 90% of its time in 10% of the code, the main slogan of this paper is: Don't worry about run-time tests in straight-line code; worry about getting efficient inner loop code.

Problems that can be alleviated by subscript analysis are starred, and are the main subject of this paper.

Out-of-bounds tests. Each element definition and each select must test whether its subscript is within array bounds. This is no different than the problem for imperative languages. We could simply turn off the out-of-bounds tests (and the collisions and empties tests) and hope for the best, as many imperative compilers do, but even for imperative languages this is a questionable compromise of semantic reliability.

For most scientific applications, subscript expressions are linear; therefore, before we enter a loop we can compute a subscript's minimum and maximum values in the loop. Therefore, the subscript bounds check can be removed to infrequently executed "front-door" or guardian code before the loop.

Collisions.* An array element may receive more than one definition. There is a cost at definition time, since each definition must test whether the element is empty (if so, set it to full) or already full (if so, flag a collision error). Subscript analysis can provide compile-time testing for collisions.

Empties. An array element may receive no definition. Every select on this array must test for an empty element. However,

> if the number of definitions = the number of elements,
>
> and there are no out-of-bounds definitions,
>
> and there are no collisions,
>
> then there can be no empties,

since the indices in the list of $(index, value)$ pairs must be a permutation of the array indices. Whether the number of definitions equals the number of elements can certainly be tested at run-time before entering a loop, and in some cases at compile-time, as can the out-of-bounds test for linear subscripts. Collisions can be detected by compile-time subscript analysis in the case of linear subscripts, so we have a complete test for empties outside of any inner loop.

Thunks.* We must create the thunks and garbage collect them when they are no longer needed. The thunks may be scattered in heap storage in a way that gives poor cache and page performance. For an array in a strict context we can partially alleviate these

creation, GC, and memory coherence problems by block allocating and deallocating the thunks, although they will still be expensive. When we select an element we must test whether the thunk has already been forced.

With safe scheduling of element evaluation we are always assured that an element's value has been evaluated and stored before we ever need it. This schedule is similar to the goal of preserving *true dependences* δ in Fortran vectorizers ([9]).

*Update copying**. For the single-element *upd* function, this is the reference-counting problem addressed by [5]. For the multi-element `bigupd` function, there are two subproblems:

(1) Don't update an array you may need later in another expression.

(2) Even if you don't need the array later, don't destroy an element's binding which you may need later in the same `bigupd` expression.

Subproblem 1 is the reference counting problem again. The difference is that for single-element *upd*, run-time reference-counting must be paid by *every* update, whereas for multi-element `bigupd` the cost of a single reference-counting operation can be amortized over *many* updates, usually over an entire row/column vector, or a large submatrix. In general, the larger the problem, the smaller the cost per element update. Even without the compile-time analysis of [5], run-time reference-counting of arrays may be an acceptably small cost. In a typed functional language, we know at compile time which expressions are arrays, so we only need to generate reference-counting code for arrays.

But even if we solve subproblem 1, subproblem 2 could still force us to copy the array. Subscript analysis of a `bigupd` can tell us that a righthand-side array select and a lefthand-side update index refer to the same element, and how to ensure that the select is scheduled before the update. This is essentially the same notion as preserving *output dependences* $\overline{\delta}$ in Fortran vectorizers ([9]).

5 Dependences detected by subscript analysis

```
a = Array ((1,1), (m,n))
    forall j <- [1..n];  [ (1,j) := 1 ]  ++      Clause C1
    forall i <- [1..m];
        [ (i,1) := 1 ]  ++                       Clause C2
        forall j <- [2..n];
            [ (i,j) := a!(i-1,j) + a!(i,j-1) ]   Clause C3
```

We can consider each clause as a template, an instance of which occurs at each point in the *iteration space* of its surrounding loops (generators). In this program, the first j loop defines a 1-D iteration space $j \in [1..n]$, at each point of which is an instance of clause C_1. Loop i defines a 1-D iteration space $i \in [1..m]$ for C_2, and the nested loop i and loop j define a 2-D iteration space $i \in [1..m]$, $j \in [2..n]$ for C_3. The iteration space for the entire expression is the union of these iteration spaces.

Deciding how to schedule these loops without compiling thunks requires knowledge of the data dependences between points of iteration space. We may even decide to perform semantics-preserving transformations that compute the same array, but use a different

iteration space, but for this paper, we will usually keep the given iteration space, and try to derive a good schedule over that space. We make some simplifying assumptions:

For sequential scheduling, a clause instance, once started, runs to completion before starting any other clause instance.

A clause is strict in every array select appearing in the *value* component expression of its (*index*, *value*) pair. In the rest of this paper, the *index* and *value* will be referred to as the clause's lefthand and righthand sides (LHS and RHS). For scientific computation this is a realistic assumption. For scheduling this is the most pessimistic assumption.

In the definition of array m, all RHS recursive selects on m are visible in the clauses; none are hidden in function calls. In this paper we do not undertake any interprocedural analysis. Hidden selects could be raised into the array comprehension by partial evaluation.

If the RHS has a recursive select that appears in a RHS loop, it must be easy to figure out the RHS loop's iteration space, e.g.:

```
b = Array (1,n)
    [ 1 := a!1 ] ++
    forall i <- [2..n];
        [ i := foldl + a!i (forall k <- [1..i-1]; b!k) ]
```

Here we make the pessimistic assumption that the element pair for $b!i$ is strict in every RHS $b!k$ for $k \in [1..i-1]$. We assume that such an expression can be efficiently compiled into DO loops without intermediate lists by Wadler's listlessness or deforestation transformations ([12,?,?]).

5.1 True dependence: loop-carried and loop-independent

Consider the sample program at the beginning of this section. We can see that C_3's RHS at some typical iteration point (x, y) uses an element computed by C_3 at an iteration points $(x-1, y)$ and $(x, y-1)$:

$$(C_3\ LHS\ a!(i,j))\ \delta(<,=)\ (C_3\ RHS\ a!(i-1,j)),$$
$$(C_3\ LHS\ a!(i,j))\ \delta(=,<)\ (C_3\ RHS\ a!(i,j-1)).$$

where
delta can be read as "supports" or "provides data for" and must therefore precede. $\delta(<,=)$ can be read as "provides data from an 'earlier' to 'later' iteration along the i dimension, and at the same iteration along the j dimension." The subscript analysis described in the last section provides this directional information, abstracting away from the actual distances. For scheduling the i and j loops without thunks, we are safe if we let both loops run from their lower to upper bounds. A short version tells us everything we need to know: $C_3\ \delta(<,=)\ C_3$, $C_3\ \delta(=,<)\ C_3$.

We can also see that C_3 at some points (x, y) depends on C_2 at point (x), where the two iteration subspaces share the i dimension, since C_2 and C_3 are surrounded by the common i loop: $C_1\ \delta()\ C_3$. Furthermore, C_3 at some points (x, y) depend on C_1 at some points (z), but we cannot provide iteration space directional information, since C_1 and C_3 are not surrounded by any shared loops: $C_2\ \delta(=)\ C_3$.

$\delta(<,=)$ and $\delta(=,<)$ are examples of *loop-carried* dependences, whereas $\delta(=)$ and $\delta()$ are examples of *loop-independent* dependences ([1]).

Without subscript analysis we would have to assume that between each pair of clauses, every possible loop-carried and loop-independent dependence may occur. With the knowledge we have here, we can see that one way to safely schedule this computation without thunks:

Complete the j loop around C_1 first, running either direction;

Run the i loop in the forward direction, being sure to complete C_2 each time before starting the inner j loop;

Run the inner j loop in the forward direction.

5.2 Antidependence

Example: left-shift vector contents one place:

```
a1 = bigupd a0
    forall k <- [1..n-1]; [ i := a0!(i+1) ]      Clause C1
```

We can see that C_1 RHS $a0!(i+1)$ at iteration space point (x) is overwritten by C_1 LHS $a1!i$ at "later" iteration $(x+1)$. We write $C_1 \ \overline{\delta}(>) \ C_1$; we can avoid copying if we run the i loop backwards. Another example:

```
a1 = bigupd a0
    forall k <- [1..n-1];
        [ i := a0!(i+1) + a1!(i-1) ]        Clause C1
```

$$C_1 \ \overline{\delta}(<) \ C_1,$$
$$C_1 \ \delta(<) \ C_1.$$

Read $\overline{\delta}$ as "is overwritten by" and must therefore precede. In a later section on antidependences, we show an example that exchanges two rows of a matrix. Our analysis can tell us how to allocate a single temporary variable to minimize copying.

5.3 Output dependence

```
a = Array (1,n)
    [ 1 := 1 ]  ++                  Clause C1
    forall j <- [1..n];
        [ j := 2 * a!(j-1) ]        Clause C2
```

We have deliberately miscoded this example: both C_1 and C_2 have definitions for $a!1$. The semantics of a sequential imperative language requires that one of multiple assignments to a memory location is defined to be the last update. Any compiler transformation of the program must preserve these semantics; therefore in imperative language, the later of two assignments to a single location is *output dependent* on the earlier assignments: the later assignment must occur after the earlier.

A functional array comprehension is single assignment. Two assignments to the same location is simply an error; discovering an output dependence between two assignments is flagged as an error. Since neither is defined to precede the other, the direction of the output dependence edge does not matter.

6 Dependence graphs

(Describe the reduced dependence graph with labeled direction vector edges derived from subscript analysis. Describe the unrolled dependence graph derived from the reduced dependence graph. Explain that any actual dependence graph generated by the array computation must be a subgraph of the unrolled dependence graph. Therefore, if analysis of the reduced dependence graph shows that a certain kind of dependence cannot hold in the unrolled dependence graph, then that kind of dependence certainly cannot occur in any actual dependence graph.)

Definition 1 *Let $G = (V_G, E_G)$ be a reduced dependence graph surrounded by a single loop $i < -[1..n]$. Construct $H = (V_H, E_H)$, the corresponding unrolled dependence graph by the following rules.*

Rule 1: Make n copies of G's nodes: $V_H = \{u_{v,i} \mid v \in V,\ i \in [1..n]\}$.

Rule 2: For each $(=) \in E$, make a corresponding edge within every iteration in H:

$$E_{H,=} = \{(u_{v,i},\ u_{w,i}) \mid (v,w) \in E,\ label(v,w) = \text{“(=)”},\ i \in [1..n]\}.$$

Rule 3: For each $(<) \in E$, make a corresponding edge every iteration i to every "later" iteration j in H:

$$E_{H,<} = \{(u_{v,i},\ u_{w,j}) \mid (v,w) \in E,\ label(v,w) = \text{“(<)”},\ i \in [1..(n-1)],\ j \in [(i+1)..n]\}.$$

Similar definition for $E_{H,>}$.

Rule 4: $E_H = E_{H,=} \cup E_{H,<} \cup E_{H,>}$.

The construction of H reflects our knowledge of the dependence direction, but our ignorance of dependence distance. Any actual dependence graph $A = (V_A, E_A)$ is a subgraph of the unrolled dependence graph $H = (V_H, E_H)$: $V_A \subseteq V_H, E_A \subseteq E_H$.

An obvious consequence of this definition:

Theorem 1 *Let G be a reduced graph, and let H be the corresponding unrolled graph. If G is acyclic, then H must be acyclic.*

Reduced dependence graph versus unrolled dependence graph. [10] refers to reduced dependence graph; what we call the unrolled dependence graph he simply calls the dependence graph. Here we will always explicityl refer to the unrolled graph; if we say merely "graph" without qualification, we mean the reduced dependence graph.

If we view each clause as a process, this could be viewed as a process scheduling problem in the special case of a single processor.

7 Dependence graphs with only true dependence edges

Output dependences have no influence on our decision whether about how to schedule execution order to avoid compiling thunks; therefore, code generation only requires us to consider true dependences and antidependences. For simplicity, we will at first consider

only the case of only the case of clauses surrounded by a single loop, then extend this strategy to multiple nested loops. We will also at first consider only dependence graphs containing true dependences (which arise with a new array), then consider graphs including both true and antidependences (which arise in *bigupd*)

Obviously, if there are no dependence edges among the clauses, there is neither loop-carried dependence across iterations nor loop-independent dependence within an iteration. In this degenerate case it does not matter in what order we execute the clauses within an iteration.

7.1 Acyclic dependence graphs

Now let us consider the case in which the dependence graph is a DAG. The next section treats a cyclic dependence graph, showing how a cyclic dependence graph can be reduced to DAG. We need to consider four kinds of acyclic graphs.

Case 1: All edges are $\delta(=)$. There are no loop-carried dependences, only loop-independent dependences within a single iteration. It does not matter what direction the loop runs. Schedule the nodes within a single iteration by topological sort.

Case 2: At least one edge is $\delta(<)$, all other edges are either $\delta(=)$ or $\delta(<)$. There is a loop-carried dependence from "earlier" to "later" iterations; all other dependences are either loop-carried in the same direction or loop-independent. The loop *must* run in the "earlier-to-later" direction. But the loop-carried edges only influence the loop direction, and have no bearing on the execution order of clauses within a single iteration. Ignore all the loop-carried $\delta(<)$ edges from the graph; the problem is now reduced to case 1 with $(<)$ loop direction enforced.

Case 3: At least one edge is $\delta(>)$, all other edges are either $\delta(=)$ or $\delta(>)$. All the loop-carried dependences are from "later" to "earlier" iterations. Reverse the specified direction of the loop; now the problem is reduced to case 2.

Case 4: At least one edge is $\delta(<)$ and at least one edge is $\delta(>)$. Apparently neither loop direction can satisfy both loop-carried dependences. However, since the dependence graph is acyclic, the unrolled dependence graph is also acyclic. We can therefore embed the acyclic unrolled graph in a total order, but we will need multiple passes through the loop.

Definition 2 *Let $G = (V, E)$ and $v, w \in V$. w is said to be $\delta(<)/\delta(=)$ supported by v if on* every *path from v to w, every edge is $\delta(<)$ or $\delta(=)$.*

There is similar definition for $v\delta(>)/\delta(=)$ supports w. An obvious theorem:

Theorem 2

(1) If there is no *path from v to w, then v both $\delta(<)/\delta(=)$ supports and $\delta(>)/\delta(=)$ supports w.*

(2) If v both $\delta(<)/\delta(=)$ supports and $\delta(>)/\delta(=)$ supports w, then either there is no path from v to w, or every edge of every path from v to w is $\delta(=)$.

Case 4 program transformation:

Let $G = (V, E)$ = the original dependence graph. Let B = set of all root nodes in V, i.e., nodes with no incoming δ edges. Since G is acyclic and non-empty, B is guaranteed to be non-empty. The nodes in B do not depend on the loop direction or any other nodes, and can be scheduled freely.

Let V_1 = the set of all nodes in V that are $\delta(<)/\delta(=)$ supported by *every* B node. Let $G_1 = (V_1, E_1)$ where E_1 is the set of all edges in E that connect nodes in V_1.

$B \subseteq V_1$, since by theorem 2, every B node $\delta(<)/\delta(=)$ supports itself and every other B node. By definition E_1 contains no $\delta(>)$ edges.

Now let $G_2 = (V_2, E_2)$, where $V_2 = V - V_1$, and $E_2 = E$ minus any edges entering or leaving V_1 nodes. Consider G_1 and G_2 as subgraphs of G.

Definition 3 *Edge $(v, w) \in E$ is said to* bridge *G_1 to G_2 if $v \in V_1$ and $w \in V_2$. Similarly, edge (v, w) bridges G_2 to G_1 if $v \in V_2$ and $w \in V_1$.*

Theorem 3 *There are no edges in E that bridge G_2 to G_1.*

Proof: Suppose the exists an edge (v, w) bridging G_2 to G_1. $v \in V_2$, so for there exists at least one $b \in B$ such that at least one path P from b to v includes a $\delta(>)$ edge. $w \in V_1$, so for every $b \in B$ no path from b to w includes a $\delta(>)$ edge. But the path P concatentated with edge (v, w) includes a $\delta(>)$ edge, which puts us in a contradiction.

Since there are no δ edges bridging G_2 to G_1, the bridging dependences will be satisfied if we wrap separate versions of the loop around each subgraph, and completely execute the G_1 loop before starting the G_2 loop. The bridging edges, which may be $\delta(<)$, $\delta(=)$, or $\delta(>)$, are no longer enclosed in a loop, and are therefore converted to loop-independent $\delta()$ edges.

Since the G_1 edges are all $\delta(<)$ or $\delta(=)$, we can schedule the loop surrounding G_1 by either case 1 or 2 above. Recursively invoke this transformation with new G = old G_2 until we reach G = a null graph. Since G in each step is finite and acyclic, B in each step is guaranteed non-empty, so G in the next step is guaranteed smaller than the previous step, and the transformation must terminate.

We can improve this transformation by maximizing the number of nodes executed in each loop pass. Let $N_<$ = the number of nodes in V that are $\delta(<)/\delta(=)$ supported by B; define $N_>$ similarly. If $N_< \geq N_>$, then let G_1 include only $\delta(<)/\delta(=)$ supported nodes and choose $(<)$ for the G_1 loop pass direction; otherwise choose $(>)$.

If we mark each G_1 graph according to the direction its surrounding loop must run to eliminate thunks, we can ignore all the loop-carried edges, leaving only $\delta(=)$ edges. Each G_1 graph is then reduced to case 1, and its nodes can be scheduled by topological sort.

Notice that cases 1-3 can be treated as simply special instances of the case 4 transformation. In these instances, the entire graph is consumed in the first and only loop pass.

7.2 Cyclic dependence graphs

We can discover cycles in a reduced dependence graph by finding the strongly connected components. By the definition of a SCC, every node v in a SCC C takes part in at least

one cycle with every other node w in C, and v takes part in no cycles with any node x not in C.

A cyclic unrolled graph H can only arise from a cyclic reduced graph G, but not every cyclic G gives rise to a cyclic H. If H is cyclic, we cannot schedule its nodes to guarantee a node is executed only after all its dependences have been satisfied; we must compile the array comprehension using thunks. But if H is acyclic, we want an algorithm to schedule the nodes so thunks are unnecessary. Looking at G, we want an algorithm that tells us whether H is acyclic, and if so, gives us a good schedule for its nodes.

Since cycles in H can only arise from cycles in G, and cycles in G can only occur in G's SCCs, we concentrate on SCCs in reduced graphs.

Definition 4 *In a reduced dependence graph:*

A $\delta(=)$ cycle contains only $\delta(=)$ edges.

A $\delta(<)$ cycle contains at least one $\delta(<)$ edge and no $\delta(>)$ edges.

A $\delta(>)$ cycle contains at least one $\delta(>)$ edge and no $\delta(<)$ edges.

A $\delta(<)/\delta(>)$ cycle contains at least one $\delta(<)$ edge and at least one $\delta(>)$ edge.

The following theorems give us a simple algorithm for determining what kind of cycles a SCC must contain and what kind it must not contain. The proofs of the first two theorems are easy.

Theorem 4 *Let $C = (V, E)$ be an SCC in a reduced graph, and let $C' = (V, E')$ where $E' = E$ after deleting all $\delta(<)$ and $\delta(>)$ edges. C contains a $\delta(=)$ cycle if and only if C' contains a cycle.*

Theorem 5 *Let $C = (V, E)$ be an SCC in a reduced graph. If E contains at least one $\delta(<)$ edge and no $\delta(>)$ edges, then C must contain a $\delta(<)$ cycle.*

This implication cannot be if-and-only-if, since E containing both $\delta(<)$ and $\delta(>)$ edges does not prohibit C containing a $\delta(<)$ cycle. An equivalent theorem holds with all δ directions reversed.

Theorem 6 *Let $C = (V, E)$ be an SCC in a reduced graph. C contains a $\delta(<)/\delta(>)$ cycle if and only if E contains at least one $\delta(<)$ edge and at least one $\delta(>)$ edge.*

Proof: The "only-if" follows immediately from the definition. The "if" follows because in an SCC, the sink vertex of every edge can reach the source vertex every other edge, therefore every pair of $\delta(<)$ and $\delta(>)$ edges must take part in a cycle.

So we have easy algorithms to establish whether SCC C has a $\delta(=)$ or $\delta(<)/\delta(=)$ cycle. We also have an easy algorithm to establish whether C has $\delta(<)$ cycle or a $\delta(<)$ cycle in the absence of a $\delta(<)/\delta(=)$ cycle. The next theorem shows that if either a $\delta(=)$ or a $\delta(<)/\delta(=)$ cycle is present, we don't care whether there are any $\delta(<)$ or $\delta(<)$ cycles.

Theorem 7 *Let C be a SCC in a reduced graph G, let H be corresponding unrolled graph for G, and let H_C be the subgraph of H whose nodes and edges were generated from nodes and edges in C.*

[1] If C contains a $\delta(=)$ cycle or a $\delta(<)/\delta(>)$ cycle, then H_C contains a cycle.

If C contains no $\delta(=)$ cycle and no $\delta(<)/\delta(>)$ cycle, then H_C cannot contain a cycle.

Proof of (1): Let C contain a $\delta(=)$ cycle; then a single iteration in H_C contains a cycle.

Now let C contain a $\delta(<)/\delta(>)$ cycle. Assume n is large compared with the number of $\delta(<)$ and $\delta(>)$ edges. Since we do not know the dependence distance of the loop-carried edges, it is possible the sums of the distances will cancel in H_C, bringing a path starting at node v in iteration i in H_C back to the same v in the same iteration i.

Proof of (2): If C contains no $\delta(=)$ cycle and no $\delta(<)/\delta(>)$ cycle, then C must contain a $\delta(<)$ or a $\delta(>)$ cycle; theorem 6 tells us C cannot contain both kinds of cycles.

Let C contain a $\delta(<)$ cycle. Choose any C node v in the cycle and any corresponding H_C node v_i in any iteration i. Follow the cycle in C, tracing one of the corresponding paths in H_C. The path in C must cross a $\delta(<)$ edge before returning to v, so every corresponding path in H_C must leave i from some "later" $j \in [(i+1)..n]$. Since there are no $\delta(>)$ edges in the C cycle, it is impossible for any H_C path to return to i.

A similar proof holds when C contains a $\delta(>)$ cycle. End of proof.

In the preceding theorem SCC C is a subgraph of reduced graph G, therefore H_C, C's unrolled graph, is a subgraph of H, G's unrolled graph. We can see that if reduced graph G contains either a $\delta(=)$ cycle or a $\delta(<)/\delta(>)$ cycle, the unrolled graph H must be cyclic, so an actual dependence graph A may be cyclic. Either every dependence in this cycle is strict, which means non-termination at run-time, or else at least one dependence is non-strict, which subscript analysis alone cannot detect at compile time. Without further information, we have no choice but to compile the array comprehension using thunks.

Likewise we can see that if reduced graph G contains no $\delta(=)$ cycle and no $\delta(<)/\delta(>)$ cycle, H and A must be acyclic; a thunkless schedule is possible.

Let us reduce each SCC C in cyclic reduced graph G to a single node C in acyclic reduced graph G'. Give C the label $\delta(=)$, $\delta(<)/\delta(>)$ or both, if it contains one of those cycles; otherwise give C one of the mutually exclusive $\delta(<)$ or $\delta(>)$ labels.

Case 1: In G' there exists a $\delta(=)$ SCC node. Compile using thunks.

Case 2: In G' there exists a $\delta(<)/\delta(>)$ SCC node. Compile using thunks.

Case 3: G' contains only $\delta(<)$ and $\delta(>)$ SCC nodes. Noncyclic edges may be $\delta(<)$, $\delta(=)$, or $\delta(>)$.

Case 3 program transformation: We can take an approach similar to the program transformation of the previous section. The following is a sketch; we will describe the transformation in more detail and prove its correctness in the final paper.

Let $G = (E,V)$ be the acyclic reduced graph with labeled SCC nodes. Let B = set of all root nodes in V, i.e., nodes with no incoming δ edges; and let $B_>$ = the set of all reduced $\delta(>)$ SCC nodes in B. Expand the $B_>$ nodes; the resulting graph can be executed in a (>) loop.

Let V_1 = the set of all nodes in V that are $\delta(<)/\delta(=)$ supported by every B node, but excluding any $\delta(>)$ labeled SCC nodes. Notice that V_1 is $\delta(<)/\delta(=)$ supported by every $B_>$ node, but that:

$$B_> \cap V_1 = \phi,$$
$$(B - B_>) \subseteq V_1.$$

The only reduced SCC nodes in V_1 are labeled $\delta(<)$. The expanded $\delta(<)$ SCC nodes and the other nodes in V_1 can be executed in a $(<)$ loop.

Finally, let $G_2 = (V_2, E_2)$, where $V_2 = V - (V_1 \cup B_>)$, and $E_2 = E$ minus any edges entering or leaving V_1 or $B_>$ nodes. Recursively invoke the transformation on a new graph $G = G_2$; terminate when G is the null graph. End of tranformation.

8 Dependence graphs with antidependence edges

Now let us consider dependence graphs in which a clause C_1 may overwrite a variable needed by clause C_2, either within an iteration or across iterations; graphs that include antidependence edges $C_1 \overline{\delta} C_2$. The function *bigupd* gives rise to such graphs.

Let us first concentrate on the simplest case of all $\delta(=)$ antidependence edges: all antidependences are within a single iteration. Recall that there is an $C_1 \overline{\delta} C_2$ edge if and only if the storage for a variable on C_1's RHS is overwritten by C_2's LHS; C_1's use of the variable's binding must precede C_2's reuse of the variable's storage.

8.1 Acyclic and cyclic loop-independent antidependences

Consider these two versions of a row exchange example from Linpack.

```
2     a1 = bigupd a0
         forall j <- [k+1..n];
           in ( [ (k,j) := a0!(1,j) ] ++      Clause C1
                [ (1,j) := a0!(k,j) ] )       Clause C2
```

$$C_2 \; \overline{\delta}(=) \; C_1,$$
$$C_1 \; \overline{\delta}(=) \; C_2.$$

```
      a1 = bigupd a0
         forall j <- [k+1..n];
           let temp = a0!(1,j);               Clause C1''
           in ( [ (1,j) := a0!(k,j) ] ++      Clause C2
                [ (k,j) := temp ] )           Clause C1'
```

$$C_1'' \; \delta(=) \; C_1',$$
$$C_1'' \; \overline{\delta}(=) \; C_2,$$
$$C_2 \; \overline{\delta}(=) \; C_1'.$$

The first program contains an antidependence cycle. The second program breaks this cycle by introducing a temporary variable that splits C_1 into two nodes C_1' and C_1'', converting the cycle to a DAG. We can break an $\overline{\delta}(=)$ antidependence cycle of *any* length by

splitting any arbitrary node in the cycle this way, which introduces a backward-pointing $\delta(=)$ edge.

How to break an $\overline{\delta}(=)$ antidependence cycle embedded in a general dependence graph: Choose an arbitrary edge $C\ \overline{\delta}(=)\ D$ in the cycle, where C's RHS variable x gets overwritten by D. Split C into nodes C' and C''. C'' is a *let* clause that binds a copy of x's value to the new variable *temp*; C' is the same as C, but with every occurrence of x on the RHS replaced by *temp*. The connectivity rules for the new graph are:

Rule 1: Introduce the true dependence $C''\ \delta(=)\ C'$.

Rule 2: C'' must copy x's binding before D destroys it, so introduce $C''\ \overline{\delta}(=)\ D$. Essentially C'' inherits the edge $C\ \overline{\delta}(=)\ D$.

Rule 3: Except for x, C' inherits all the RHS variables of C, therefore C' inherits all $C\ \overline{\delta}(*)\ Y$ edges for all nodes $Y \neq D$, and all $Y\ \delta(*)\ C$ edges for all nodes Y. Here $(*)$ means C' inherits an edge whatever its directionality.

Rule 4: C' writes the same LHS variable as C, therefore C' inherits all $C\ \delta(*)\ Y$ edges and all $Y\ \overline{\delta}(*)\ C$ edges for all nodes Y.

Notice this algorithm for antidependence cycle-breaking assumes the only occurrences of x that can be reached by executing clause C appear explicitly on C's RHS. This has been our assumption throughout the paper; could be ensured by searching call graph, or by lambda-lifting.

Deciding the execution order of clauses in graphs involving loop-independent antidependences now comes down two cases. Notice that $\overline{\delta}(=)$ cycles can never prevent thunk-free compilation.

Case 1: All $\overline{\delta}$ edges are $(=)$ and there are no $\overline{\delta}(=)$ cycles. Then treat the $\overline{\delta}(=)$ edges the same as $\delta(=)$ edges; the problem is reduced to the the case of a general δ graph treated above.

Case 2: All $\overline{\delta}$ edges are $(=)$ and there is at least one $\overline{\delta}(=)$ cycle. Choose any $\overline{\delta}(=)$ cycle, then break the cycle by splitting any node C in that cycle. Repeat until all the $\overline{\delta}(=)$ cycles are broken. The problem is now reduced to case 1.

So we can always break $\overline{\delta}(=)$ cycles; $\delta(=)$ cycles require us to compile thunks, as described above. But what about a cycle that has at least one each of both $\overline{\delta}(=)$ and $\delta(=)$ edges?

Redefine an $\overline{\delta}(=)$ cycle to be any cycle that contains only $\overline{\delta}(=)$ and $\delta(=)$ edges, at least one which is $\overline{\delta}(=)$. If we choose any node C that contributes an outgoing $\overline{\delta}(=)$ edge to the cycle, we can always break the cycle by the same node-splitting algorithm described above; essentially we are prefetching the variable binding that causes the outgoing $\overline{\delta}(=)$ edge. Case 2 now applies using the new definitions of $\overline{\delta}(=)$ cycles and of $\overline{\delta}(=)$ cycle-breaking.

8.2 Acyclic and cyclic loop-carried antidependences

At first we will consider only graphs that include loop-carried true dependences in only the $\delta(<)$ direction. Graphs that include only $\delta(>)$ loop-carried true dependences are easily converted to $\delta(<)$ by reversing the loop direction. Graphs that include loop-carried true dependences in ***both*** directions are an easy variation on the $\delta(<)$ single-direction case.

Case 1: There is at least one $\overline{\delta}(<)$ edge; all δ edges and all other $\overline{\delta}$ edges are either $(<)$ or $(=)$. If there are any $\overline{\delta}(=)$ cycles, break them as described above. Now any $\overline{\delta}$ cycles left

contain at least one $\overline{\delta}(<)$ edge that leads to a later iteration; there is no δ or $\overline{\delta}$ edge that can lead back to the earlier iteration; so the unrolled dependence graph cannot contain any cycles involving $\overline{\delta}$ edges (although $\delta(=)$ cycles are still possible). All the loop-carried $\overline{\delta}(<)$ and $\delta(<)$ edges agree on the loop direction; treat all $\overline{\delta}$ edges as if they are δ, reducing the problem to the case of a general δ graph.

Case 2: There is at least one $\overline{\delta}(>)$ edge; all δ edges are either $(<)$ or $(=)$; all other $\overline{\delta}$ edges are either $(<)$, $(>)$ or $(=)$. Two subcases:

Case 2.1: There are only $\delta(=)$ edges, no $\delta(<)$ edges. The loop direction is only constrained by the $\overline{\delta}(>)$ edge's need to use bindings before they are destroyed; there is no true data flow dependence. However, there may be conflicting $\overline{\delta}(<)$ constraints.

Case 2.2: There are $\delta(<)$ edges, which conflict with the $\overline{\delta}(>)$ constraint.

If we can remove all the $\overline{\delta}(>)$ edges, we can then reduce case 2 to the purely $\overline{\delta}(<)$ case 1 above. We do this by a variation on the node-splitting algorithm described for breaking $\overline{\delta}(=)$ cycles, but here we introduce a *vector* of temporary variables. Let C $\overline{\delta}(>)$ D be the offending edge. C uses variable x's binding in a "later" iteration, D destroys the binding in an "earlier" iteration. The following is only a sketch of the transformation; details will be provided in the final paper.

Rule 1: Split node C into nodes C' and C". Introduce the true dependence C" $\delta(<)$ C'. Essentially we have kept C', which uses x's binding, in the same "later" iteration in which C was using x, but we have move C", which copies x's binding to a temporary, into the same "earlier" iteration with D.

Rule 2: C" must copy x's binding before D destroys it, so introduce C" $\overline{\delta}(=)$ D. Essentially C" inherits the edge C $\overline{\delta}(>)$ D, but in the "earlier" iteration with D.

Rules 3 and 4: Same as $\overline{\delta}(=)$ node splitting.

Essentially we are copying the old array into a temporary array before updating the old array to get the new array, just as a naive implementation of *bigupd* would copy the old array into a new array and then update the new array. But there are advantages to our approach. For example, we may only need to copy a single row or column vector from a 2-D array instead of the entire array, and perhaps only part of the vector at that. If the "distance" between the loop iteration that destroys a binding and the iteration that uses the binding is small, we could use a circular buffer to hold temporarily the old bindings; if the distance is very small, let us say only 1 or 2 iterations, the compiler may be able to implement this circular buffer efficiently in fast floating point registers.

However, getting the details right with this optimization is fraught with complications. This transformation sketch for case 2 will be discussed in more detail in the final paper.

Case 3: There is at least one of loop-carried $\overline{\delta}$ edge (either $\overline{\delta}(<)$ and $\overline{\delta}(>)$); and there are loop-carried δ edges in *both* directions (both $\delta(<)$ and $\delta(>)$). We can certainly eliminate the loop-carried $\overline{\delta}$ edge by node splitting, but the conflicting δ edges still don't permit a safe direction for the loop, so we must use thunks at least for the new array. Further discussion in the final paper.

9 Multiple nested loops

Multiple nested loops will be treated in detail in a future paper; here we will confine ourselves to some general remarks.

Multiple nested loops surrounding an *acyclic* reduced dependence graph can be treated as a generalization of a single loop surrounding an acyclic dependence graph. Consider the case of two nested loops with loop index variables i and j. Let us visualize the unrolled dependence graph as a matrix in which i indexes rows and j index columns.

When deciding the direction for the outer i loop, only the $(<,*)$ and $(>,*)$ edges influence the ordering of rows. The $(=,*)$ edges can be ignored since in the unrolled graph they only describe edges *within* a row. We may need to apply some of the antidependence cycle-breaking and loop-splitting transformations to the graph G, obtaining a new graph G'. But then G' is treated as the inner j loop's new graph; the outer i loop will treat the entire inner j loop as a single node graph.

Now decide the direction for the inner j loop. We can now eliminate the $(<,*)$ and $(>,*)$ edges, since they reach across rows; only $(=,*)$ edges influence ordering within a row. For any arbitrary number of loops $i_1,\ldots,i_d$ surrounding an acyclic dependence graph, when we consider loop k we only look at edges

$$(=_1,\ldots,=_{k-1},i_k,*_{k+1},\ldots,*_d)$$

where i_k is either $<$ or $>$; and as we pass inward to consider loop $k+1$, we eliminate these edges, retaining only edges for which i_k is $=$.

Once we have set the directions for all the loops, we are left with in problem of ordering the nodes within a single iteration. At this point we are left only with $(=_1,\ldots,=_d)$ edges, just as we would expect.

Demonstrating that a multiloop *cyclic* reduced dependence graph cannot give rise to a dependence cycle in the unrolled graph is more difficult than the same analysis of a single-loop cyclic reduced graph. This problem will be treated in a future paper, but we can show some of the difficulties in an example. A graph with a cycle that includes both $(<,=)$ and $(>,=)$ edges may result in an unrolled graph with a cycle. So may a graph with a cycle that includes $(=,<)$ and $(=,>)$ edges. But we must also watch for graph cycles in which the paths such as

$$(=,<) \rightarrow (<,=) \rightarrow (=,>) \rightarrow (>,=)$$

are possible. In the unrolled graph we could read this path as "right $\rightarrow$ down $\rightarrow$ left $\rightarrow$ up," which may give rise to a cycle in the unrolled graph. This case also may *not* give rise to an unrolled graph cycle, but we cannot detect this with only dependence direction information.

Unfortunately, the obvious generalization of the SCC labeling algorithm from the single loop to the multiple loop case is too conservative; it may flag cycles that cannot possibly give rise to unrolled dependence graph cycles, even based only on dependence direction information.

10 Subscript analysis

We will briefly outline subscript analysis in the case of one-dimensional subscripts. Consider two references to array v, both surrounded by d shared loops.

```
Array (. . .)
 forall i1 <- [1..N1];
    . . .
      forall id <- [1..Nd];
         [...m!(f i1 ... id)...
          ...m!(g i1 ... id)...]
```

We want to know if there should be a dependence edge δ from reference $m!(f\ i_1\ \ldots\ i_d)$ to reference $m!(g\ i_1\ \ldots\ i_d)$. We use δ here to represent any kind of dependence edge, δ, $\bar{\delta}$, or δ°; the particular kind depends upon the context and is not relevant here.

Definition 5 *The bounded integer solution test:* $m!(f\ i_1\ \ldots\ i_d)\ \delta\ m!(g\ i_1\ \ldots\ i_d)$ *if and only if within the region of interest*

$$R = \{(x_1,\ \ldots,\ x_d,\ y_1,\ \ldots,\ y_d) \mid x_1, y_1 \in [1..N_1],\ \ldots,\ x_d, y_d \in [d..N_d]\}$$

there exists an integer solution to the dependence equation

$$h\ x_1\ \ldots\ x_d\ y_1\ \ldots\ y_d\ =\ (f\ x_1\ \ldots\ x_d) - (g\ y_1\ \ldots\ y_d)\ =\ 0.$$

We also want to label the dependence edge with a constraint on each loop, each of which represents a constraint on R. For example, $m!(f\ i_1\ i_2\ i_3\ i_4)\ \delta(=,<,>,*)\ m!(g\ i_1\ i_2\ i_3\ i_4)$ means a dependence holds with the constraints $x_1 = y_1$, $x_2 < y_2$, $x_3 > y_3$, no constraint on the values of x_4 and y_4. The list of constraints $(?_1,\ldots,?_d)$, where $?_k \in \{*,<,=,>\}$, is called a *direction vector.*

There are tractable decision algorithms when the subscript functions are linear in the loop variables:

$$f\ x_1\ \ldots\ x_d\ =\ a_0 + \textstyle\sum_{k=1}^{d} a_k x_k,$$
$$g\ y_1\ \ldots\ y_d\ =\ b_0 + \textstyle\sum_{k=1}^{d} b_k y_k.$$

Then the dependence equation becomes

$$\sum_{k=1}^{d} a_k x_k\ -\ \sum_{k=1}^{d} b_k y_k\ =\ b_0 - a_0.$$

The d nested loops are labeled from the set $Q = [1..d]$. Partition Q according to the constraints placed on x_k and y_k by region R:

$$Q_* = \{k | k \in Q \text{ and } ?_k \text{ is } \text{“}*\text{”}\},$$

with similar definitions for $Q_<$, $Q_=$, and $Q_>$. We can then write the dependence equation

$$\sum_{k \in Q_=} (a_k - b_k) x_k\ +\ \sum_{k \in (Q - Q_=)} a_k x_k\ -\ \sum_{k \in (Q - Q_=)} b_k y_k\ =\ b_0 - a_0.$$

Linear diophantine equation theory gives us an exact test (necessary and sufficient), but it is exponential in the number of surrounding loops ([14], [6]). For 1 or 2 levels of nesting this algorithm is reasonable. But we can derive two inexact tests (necessary but not sufficient) that are linear in the number of loops. There is a dependence only if there is an integer solution to the dependence equation, regardless of whether the solution falls within R.

Theorem 8 *The any integer solution test:* $m!(f\ i_1\ \ldots\ i_d)\ \ \delta\ \ m!(g\ i_1\ \ldots\ i_d)$ *only if*

$$\gcd(\ldots, a_j - b_j, \ldots, a_k, \ldots, b_k \ldots) \mid b_0 - a_0$$
$$\text{where } j \in Q_=$$
$$\text{and } k \in (Q_< \cup Q_> \cup Q_*).$$

And there is a dependence only if there is a rational solution to the dependence equation within R.

Theorem 9 *The bounded rational solution test: If* $(h\ x_1\ \ldots\ x_d\ y_1\ \ldots\ y_d)$ *is continuous, then* $m!(f\ i_1\ \ldots\ i_d)\ \ \delta(?_1, \ldots, ?_d)\ \ m!(g\ i_1\ \ldots\ i_d)$ *only if*

$$\min_R(h\ x_1\ \ldots\ x_d\ y_1\ \ldots\ y_d) \leq 0 \leq \max_R(h\ x_1\ \ldots\ x_d\ y_1\ \ldots\ y_d)$$

This last theorem is developed into the Banerjee test. The GCD test is cheaper than either the exact test or the Banerjee test, and is therefore applied first. If a pair of references satisfies the GCD test, try to disprove the dependence with the Banerjee test; and if we cannot disprove it, find the direction vector for the dependence from the Banerjee test.

Definition 6 *If* t *denotes a real number, then the* positive part t^+ *and the* negative part t^- *of* t *are defined as:*

$$\begin{aligned} t^+ = &\ \ t, && \text{if } t \geq 0, \\ &\ \ 0, && \text{if } t < 0. \\ t^- = &\ -t, && \text{if } t \leq 0, \\ &\ \ 0, && \text{if } t > 0. \end{aligned}$$

Let $k \in Q_*$. By an easy variation of the proof of Theorem 4 in [1], we can find the minimum and maximum for the term $a_k x_k - b_k y_k$, where the relative values of x_k and y_k are unconstrained in region R:

$$(a_k - b_k) - (a_k^- + b_k^+)(M_k - 1) \ \leq\ a_k x_k - b_k y_k \ \leq\ (a_k - b_k) + (a_k^+ + b_k^-)(M_k - 1)$$

Similar minima and maxima can be derived for the cases $k \in Q_<, Q_=$, and $Q_>$ in which region R constrains the relative values of x_k and y_k (see [1], [14]). Gathering these inequalities we get:

$$\begin{aligned} \min_R h = &\ \sum_{k=0}^{d}(a_k - b_k) \\ &+ \sum_{k\in Q_*} -(a_k^- + b_k^+)(M_k - 1) \quad + \sum_{k\in Q_<}(-b_k - (a_k^- + b_k)^+(M_k - 2)) \\ &+ \sum_{k\in Q_=}(-(a_k - b_k)^+(M_k - 1)) \ + \sum_{k\in Q_>}(a_k - (a_k + b_k^+)^-(M_k - 2)) \\ &\leq 0 \leq \\ \max_R h = &\ \sum_{k=0}^{d}(a_k - b_k) \\ &+ \sum_{k\in Q_*}(a_k^+ + b_k^-)(M_k - 1) \quad + \sum_{k\in Q_<}(-b_k + (a_k^+ - b_k)^+(M_k - 2)) \\ &+ \sum_{k\in Q_=}(-(a_k - b_k)^+(M_k - 1)) \ + \sum_{k\in Q_>}(a_k + (a_k + b_k^-)^+(M_k - 2)) \end{aligned}$$

This inequality can be checked for any one direction vector in $O(d)$ time; there are 3^d fully constrained direction vectors (direction vectors for which no $?_k = *$, therefore $Q_* = \phi$).

[6] suggests a search tree approach. Let the search tree root be $\delta(*_1, \ldots, *_d)$. The Banerjee inequality poses a necessary test, so if it cannot be satisfied over this unconstrained region R, it certainly cannot be satisfied over a more constrained region.

But if it can be satisfied, expand the root node into three children, each presenting an added constraint on R; e.g., $\delta(<_1, *_2, \ldots, *_d)$, $\delta(=_1, *_2, \ldots, *_d)$, and $\delta(>_1, *_2, \ldots, *_d)$. Each new Banerjee inequality can be computed from the parent node's inequality in constant time. If any given node fails the Banerjee inequality, there is no need to search the tree rooted at that node. The search time can be as bad as $d + (3^d - 1)/2)$, which although exponential, is still better than $d\ 3^d$, but it can be as good as $d + 3d$ when a dependence exists, or even d when there is no dependence.

10.1 Unshared loops

We have developed this for the case in which both array references are surrounded by d common loops. But within the d shared loops, the references a and b may be further surrounded by respectively d_a or d_b unshared loops. Arbitrarily label these loops:

$$k \in Q_a = [(d+1)..(d+d_a)],$$
$$k \in Q_b = [(d+d_a+1)..(d+d_a+d_b)].$$

In the following example, $d = 2$, $d_a = 1$, and $d_b = 1$:

```
m = Array (. . .)
    forall i1 <- [1..N1];
        forall i2 <- [1..N2];
            (forall i3 <- [1..N3];
                [...m!(a0 + a1*i1 + a2*i2 + a3*i3)...]) ++
            (forall i4 <- [1..N4];
                [...m!(b0 + b1*i1 + b2*i2 + b4*i4)...]);
```

Now the dependence equation becomes:

$$(a_0 + \sum_{k \in (Q \cup Q_a)} a_k x_k) - (b_0 + \sum_{k \in (Q \cup Q_b)} b_k x_k) = 0$$

For an unshared loop $k \in Q_a$, the value x_k has no corresponding value y_k, so we need to find the minimum and maximum of the term $a_k x_k$ (again, see the proof of Theorem 4 in [1]):

$$a_k - a_k^-(M_k - 1) \quad \leq \quad a_k x_k \quad \leq \quad a_k + a_k^+(M_k - 1)$$

For an unshared loop $k \in Q_b$:

$$b_k + b_k^+(M_k - 1) \quad \leq \quad b_k y_k \quad \leq \quad b_k + b_k^-(M_k - 1)$$

For every $k \in Q_a$ and every $k \in Q_b$, add these terms into the Banerjee inequality developed in the previous section. Since x_k for $k \in Q_a$ cannot be constrained relative to any y_k for $k \in Q_b$, the direction vector has components only for the d shared loops.

References

[1] R. Allen and K. Kennedy. Automatic translation of fortran programs to vector form. *ACM TOPLAS*, 9(4):491–542, 1987.

[2] S. Anderson. *Efficient Compilation of Functional Arrays.* Technical Report, Yale, 1989.

[3] Utpal Banerjee. *Speedup of Ordinary Programs.* PhD thesis, University of Illinois, 1979.

[4] A. Bloss and P. Hudak. Path analysis. In , page , 1988.

[5] M. Burke and R. Cytron. Interprocedural dependence analysis and parallelization. In *ACM POPL*, pages 162–175, 1986.

[6] et al. Hudak, P. *Report on the Functional Programming Language Haskell: Draft Proposed Standard.* Technical Report YALEU/DCS/TR-666, Yale University, December 1988.

[7] P. Hudak. A semantic model of reference counting. In *ACM Conf.Lisp and Functional Programming*, pages 351–363, 1986.

[8] D.A. Padua and M.J.Wolfe. Advanced compiler optimizations for supercomputers. *CACM*, 29(12):1184–1201, 1986.

[9] S. Rao and Kailath. Regular iterative algorithms. In E.E. Swartzlander, editor, *Systolic Signal Processing Systems*, M.Dekker, 1987.

[10] D.J.Kuck U. Banerjee, S.C.Chen and R.A.Towle. Time and parallel processor bounds for fortran-like loops. *IEEE Trans.Computers*, C-28(9):660–670, September 1979.

[11] P. Wadler. List comprehensions. In S.L. Peyton Jones, editor, *Implementation of Functional Programming Languages*, Prentice-Hall, 1987.

[12] P. Wadler. Listlessness is better than laziness: lazy evaluation and garbage collection at compile time. In *ACM Proc.Lisp and Functional Programming, Austin*, pages 45–52, 1984.

[13] P. Wadler. A new array operation for functional languages. In *LNCS 295: Proc.Graph Reduction Workshop, Santa Fe*, page , Springer-Verlag, 1986.

[14] Michael J. Wolfe. *Optimizing Supercompilers for Supercomputers.* PhD thesis, University of Illinois, 1982.