Compilation of Haskell Array Comprehensions for Scientific Computing

Steven Anderson Paul Hudak *

Yale University
Department of Computer Science
Box 2158 Yale Station
New Haven CT 06520-2158
anderson-steve@cs.yale.edu hudak-paul@cs.yale.edu

Abstract

Monolithic approaches to functional language arrays, such as Haskell *array comprehensions*, define elements all at once, at the time the array is created, instead of incrementally. Although monolithic arrays are elegant, a naive implementation can be very inefficient. For example, if a compiler does not know whether an element has zero or many definitions, it must compile runtime tests. If a compiler does not know inter-element data dependences, it must resort to pessimistic strategies such as compiling elements as thunks, or making unnecessary copies when updating an array.

Subscript analysis, originally developed for imperative language vectorizing and parallelizing compilers, can be adapted to provide a functional language compiler with the information needed for efficient compilation of monolithic arrays. Our contribution is to develop the number-theoretic basis of subscript analysis with assumptions appropriate to functional arrays, detail the kinds of dependence information subscript analysis can uncover, and apply that dependence information to sequential efficient compilation of functional arrays.

1 Introduction

In recent years many proposals have been put forth for incorporating arrays into functional languages, the differences being captured in trade-offs between expressiveness and efficiency. One of the most popular proposals puts forth what are called *non-strict monolithic arrays*. Versions of such arrays have been used in several functional languages, including FEL [17], Alfl [9], and Id-Nouveau [19], and a number of articles have been written about them [22, 10]. Most recently, a form of such arrays has been adopted in Haskell [14], where they are called "array comprehensions."

Non-strict arrays may contain undefined (i.e., $\perp$) elements, yet still be well-defined overall; this in contrast to *strict* arrays which are completely undefined if any one element is undefined. Non-strict arrays are more in the spirit of lazy, or call-by-need evaluation, whereas strict arrays capture the essence of call-by-value computation.

Monolithic arrays are ones whose elements are defined all at once, at the moment the array value is created, in contrast to *incremental* arrays whose values are defined incrementally. Monolithic arrays are more in the spirit of functional programming, whereas incremental arrays, although functional, resemble the imperative array model. There are many variations of monolithic arrays (four different kinds are described in [22]), and aside from the non-strict arrays found in the languages mentioned previously, strict variations are used in VAL [18], Connection Machine Lisp [20], and of course, APL [16].

Scientific computing makes extremely stringent demands on run-time efficiency. Although non-strict monolithic functional language arrays, such as Haskell array comprehensions, are semantically elegant and expressive, a naive implementation is prohibitive for scientific computing. In this paper we focus on techniques for the efficient compilation of non-strict monolithic arrays for sequential execution. We will use a slight generalization of Haskell syntax throughout, although our ideas may be easily applied to other languages. We adapt subscript analysis, originally developed for imperative language compilers, to the problems encountered in compiling functional language arrays. We also introduce several syntactic extensions to Haskell that retain expressiveness while conveying useful information to the optimizing phases of the compiler.

Imperative languages *overspecify* evaluation order of array elements, whereas functional languages *underspecify* evaluation order. Compilers for both make the pessimistic assumption of overestimating dependencies between array elements. Both therefore resort to pessimistic strategies to preserve semantics: imperative languages prohibit vectorization and parallelization, whereas functional languages use expensive run-time representations, even for sequential execution. Subscript analysis, by giving a better approximation to the true partial order of dependences, permits important optimizations.

We begin by discussing the advantages of non-strict monolithic arrays evaluated in a strict context for achieving both expressiveness and efficiency. We then introduce Haskell array comprehensions, and an extension called nested comprehensions that offer more control over the order in which

*This research was supported by the Department of Energy under grant DE-FG02-86ER-25012

Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation. White Plains, New York, June 20-22, 1990.

array elements are specified. We detail the possible run-time inefficiencies of functional language arrays, and the compiler analyses needed to remove these inefficiencies. We outline the number theoretic basis of the subscript analysis needed to create the dependence graphs to support some of these optimizations. We present efficient algorithms on these dependence graphs that statically schedule sequential computation of monolithic arrays and of incremental arrays to remove these inefficiencies. In conclusion, although we focus on sequential computation, we suggest how this work can be applied to vectorization and parallelization of functional language programs.

2 Non-Strict versus Strict Arrays

We will define the difference between non-strict and strict arrays, and the advantages and drawbacks of each. We will also define the idea of evaluating a non-strict array in a strict context, which combines the advantages of both strict and non-strict arrays.

There are many ways to consider arrays, but for the present we consider an array a constructor, similar to CONS, but with an arbitrarily large number of element slots. Because the number of elements can be arbitrary, we do not provide a separate selector for each element, but instead a single selector that takes both the array and an element subscript as its arguments.

[22] considers several different ways to define a *monolithic array*, i.e., an array whose elements are completely defined by the call to the constructor function that originally creates the array. Let *array* be such a constructor function. We will discuss the semantics of *array* in the next section; but for now we can distinguish strict and non-strict versions of the constructor function. We will call an array *strict* if it is returned by a strict constructor function, which evaluates all the elements of the monolithic array at its creation, and *non-strict* or *lazy* if it is created by a non-strict constructor function, which defines the elements of the array but does not evaluate them.

Definition: Let $a!i$ be the value at subscript i of array a. An array a is *strict* if for any i within the array bounds of a, $a!i = \perp$ implies that $a = \perp$.

It is easy to show that if a strict array is recursively defined, the entire array must evaluate to $\perp$ ([3]). In fact the computation of any recursively defined data structure built with strict constructor functions is always nonterminating.

Because the mathematical specification of arrays in scientific computing so often uses recurrence equations, we prefer a non-strict array constructor for its expressiveness. Unfortunately, non-strict arrays in general must represent the delayed computation of array elements as closures, or thunks, which have significant runtime costs. These costs will be detailed in a later section.

But in nearly all scientific programs, the programmer knows that an array is used in a context that will demand all the elements. We say that an array is *used in a strict context* if the surrounding program is guaranteed to demand the value of every element in the array. If we know:

- that a non-strict array is used in a strict context, and
- a safe (partial) order of evaluation of the elements, such that no element is evaluated until *after* every element on which it has a dependence,

then we perhaps have the opportunity to treat the non-strict array operationally like a strict array, storing and selecting element values directly rather than as closures. Finding a safe schedule is the subject of the latter part of the paper. But how to be sure the array is used in a strict context?

There is an extensive literature in the functional language community on analyzing the strictness of a program using non-strict data structures such as a list. We could try to develop a compile-time strictness analysis for arrays. Such an analysis may be an interesting direction for further research, and will certainly need to incorporate a subscript analysis similar to the one presented here, or an analysis of simple sections such as [4]. The drawback is that such an analysis is likely to be intellectually complex and computationally expensive, since it will generally need to analyze not just the recursive context in which the array is defined, but the entire program in which it is used. This is unfortunate, since the scientific programmer nearly always intends that all the array elements be evaluated when it is defined.

Indeed, discovering the pattern of recursive dependences may require analysis over the entire program, not just the local definition of the array. Consider the function:

```
f u = letrec v = ... u ...
      in v
```

where u and v are both non-strict arrays. The definition of v depends upon u, but apparently is not self-dependent. Or is it? Consider the context `letrec a = g (f a)` where a is a non-strict array. Clearly this program context makes v's definition recursive. A scientific programmer rarely expresses such an obscure form of self-dependence, but the compiler (or another programmer) can never be sure by inspecting only the definition of f.

We will take the path of letting the programmer specify that an array is used in a strict context. Let us define a function `force-elements`, which forces a demand for *every* element of its array argument, returning a "strictified" version of the array;

$$(\mathit{force\text{-}elements}\ a)!i = \begin{cases} \perp & \text{if } (\exists j) : a!j = \perp \\ a!i & \text{otherwise} \end{cases}$$

If we recursively define a non-strict array a, such that the recursive definition only refers to a, but all other references are to `a' = force-elements a`, then we can be sure that a is used in a strict context. We introduce as syntax a new version of `letrec` to define bindings for non-strict arrays in a strict context:

```
(letrec* x = E0 in E1)  =
    (\ x . E1) (force-elements (fix (\ x . E0)))
```

Naturally, `letrec*` can introduce multiple mutually recursive bindings by treating x as a tuple.

If we rewrite our example as `f u = letrec* v = ... in v`, then we can be sure that every element of v is evaluated before v is returned from f. When the elements of v are

of a type with only zero-ary constructors, such as floating point numbers, we can be sure every element is *completely* evaluated. In this case, if a caller to `f` creates a hidden recursive dependence such as `letrec a = g (f a)`, then `v` = ⊥. We can be confident that if `v` has any recursive dependence on itself, this dependence will appear explicitly in the `letrec*` that defines `v`.

`letrec*` allows the programmer to specify easily and elegantly both that an array is recursively defined and that every element of an array is evaluated before the array is used. It also specifies that there can be no subtle hidden recursive dependences outside the scope of the `letrec*` bindings.

3 Haskell's Array Comprehensions

Haskell has a family of non-strict monolithic arrays whose special interaction with list comprehensions provides a convenient "array comprehension" syntax for defining arrays monolithically. As an example, here is how to define a vector of squares of the integers from 1 to n:

```
let a = array (1,n)
      [ (i,i*i) | i <- [1..n] ]
```

The first argument to `array` is a tuple of *bounds*, and thus this array has size n and is indexed from 1 to n. The second argument is a list of subscript/value pairs whose order is irrelevant, but which can only have one value associated with each subscript. It is written here as a conventional "list comprehension," which resembles mathematical set notation and has become a popular syntax for lists within the functional programming community.

The `i`th element of an array `a` is written `a!i`, and thus in the above case we have that `a!i` = `i*i`.

We introduce the binary infix operator `:=`, which returns the pair of its two arguments, as syntax to reduce the number of parentheses and make array comprehensions more readable. This syntax is meant to be reminiscent of the equal sign in a *let* binding: read `i := v` as "let array element `i` equal `v`."

Perhaps the nicest aspect of non-strict monolithic arrays is that one can program in a style that not only resembles mathematical notation, but that also preserves the mathematical semantics. In particular, they allow the expression of *recursive* arrays in a concise, perspicuous, and well-defined manner. For example, consider this mathematical specification of an array a:

$$a_{i,j} = \begin{cases} 1 & \text{if } i = 1,\ 1 \le j \le n \\ 1 & \text{if } j = 1,\ 2 \le i \le n \\ a_{i-1,j} + a_{i-1,j-1} + a_{i,j-1} & \\ \quad 2 \le i \le n,\ 2 \le j \le n & \end{cases}$$

This is a specification of an $n \times n$ matrix using a "wavefront recurrence" where the north and west borders are 1, and each other element is the sum of its north, north-west, and west neighbors. In Haskell this would be written (using `++` as the infix operator for append):

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

Note the similarity between the two specifications. Although the subscript/value pairs are expressed as a list, the order of the list is completely irrelevant, thanks to the non-strict semantics of the array. In implementing such recurrences in, say, Fortran, one must be sure to write the program such that the elements are stored in an order consistent with the dependencies. With non-strict arrays, one is freed of that concern.

So far our monolithic array function assumes that each possible subscript is specified exactly once. Haskell also offers a more general monolithic array function that relaxes this requirement [15]. An *accumulated array* is created by specifying a default element value `a` for an element that receives no definitions, and a combining function `f` to combine the values for an element that receives multiple definitions. If `f` is not associative and commutative, the order of `svpairs` must be preserved. The analysis we develop in later sections apply to ordinary monolithic arrays in which elements with no definitions or multiple definitions are treated as errors, and therefore the order of `svpairs` can be changed. An interesting direction for further work would be to extend this analysis to general accumulated arrays.

3.1 Generalization of List Comprehensions

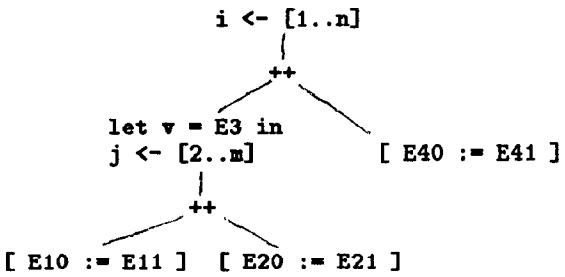
In our experience using array comprehensions in scientific computation, we have found it convenient to use a slight generalization of the list comprehension syntax that provides more control over the creation of intermediate lists; this has the added benefit of allowing us to state more clearly the optimizations that we perform. These *nested* list comprehensions, as we call them, look very much like ordinary list comprehensions, but allow the specification of a tree-shaped hierarchy of lists, all of which are appended together to yield the final result. For example, consider this ordinary list comprehension:

```
a = array ((1,1), (m,n))
      [ E10 := E11 where v = E3
        | j <- [2..m], i <- [1..n] ]
   ++ [ E20 := E21 where v = E3
        | j <- [2..m], i <- [1..n] ]
   ++ [ E40 := E41 | i <- [1..n] ]
```

Ordinary list comprehensions offer no way to express the notion that all the expressions are nested within the shared generator `i <- [1..n]`, that `E10 := E11` and `E20 := E21` are further nested within shared generator `j <- [1..m]`, that `v` is a free variable over both E1 and E2 binding common subexpression E3, and that `i` but not `j` is free over E3. Using nested list comprehensions (written with `[* ... *]` brackets), we could write instead:

```
a = array ((1,1),(n,n))
    [* ( [* [ E10 := E11, E20 := E21 ]
            | j <- [2..m] *]
       where v = E3) ++ [ E40 := E41 ]
     | i <- [1..n] *]
```

Consider the expression tree for this nested list comprehension,



Each node returns a list. The append nodes permit branching into different kinds of list expressions. The generator nodes, such as `i <- [1..n]`, create one instance of the list below it for each instance of the generator index `i`, then append those instances together to yield a result list.

Although the order of the list of subscript/value pairs is irrelevant to the semantics of an array comprehension, it has a great effect on the efficiency, and thus we would like to express the order that best suits our needs; nested array comprehension give us just that degree of expressiveness. In the remainder of this paper we will use nested list comprehensions.

A nested list comprehension is a only a syntactic extension that is translated by a translation rule TE to more primitive language constructs. TE for nested comprehensions is a simple extension of TE for ordinary list comprehensions given in [23]:

```
TE{ [* E | i <- L *] }
          = flatmap (\ i . TE{ E }) L
TE{ [* E | i <- L; Q *] }
    = flatmap (\ i . TE{ [* E | Q *] }) L
TE{ [* E | B *] }
          = if B then TE{ E } else []
TE{ E1 ++ E2} }
          =  TE{ E1 } ++ TE{ E2 }
TE{ let BINDS in E }
          =  let BINDS in TE{ E }
TE{ [ E ] }
           =  [ E ]

flatmap f []   = []
flatmap f x:xs = (f x) ++ (flatmap f xs)
```

We include rules for *let* constructs and the append operator ++, since they give the flexibility that ordinary list comprehensions lack. The only real change is in the first rule, which for ordinary comprehensions is:

```
TE{ [ E  | i <- L ] }
       = flatmap (\ i . TE{ [ E ] }) L
```

TE makes the semantics of nested comprehensions clear, but as an implementation it requires a tremendous amount of unnecessary CONSing. [23] transforms the simple translation TE for ordinary comprehensions into a more complicated translation that creates no CONS cell that is not part of the result; this transformation can be easily adapted to TE for nested comprehensions.

But we can go further. The vast majority of scientific applications can be expressed as `foldl` of some operator over a list, where:

```
foldl f a []   = a
foldl f a x:xs = foldl f (f a x) xs
```

And the vast majority of scientific applications can be expressed as list comprehensions in which the generators are over an arithmetic sequence of integers, such as:

```
i <- [low,inc..high]
i <- [high,dec..low]
```

For example:

```
sum [ a!k * b!k | i <- [1..n] ]
where sum xs = foldl (+) 0. xs
```

We can *always* transform this pattern, the application of `foldl` to a list comprehension over arithmetic sequence generators, as syntax, we can *always* into the application of a specialized first-order tail-recursive function that create no CONS cells – no intermediate lists – whatsoever. In other words, we translate such `foldl` calls into DO loops, where the generators become loop indices, and the accumulator argument a in `foldl` holds the 'state' updated by the loop. See [3] for formal details. This pattern is so common that we propose to treat it as syntax, just as we treat list comprehensions as syntax.

Since it is often clearer to express a notion such as `sum` by encapsulating the `foldl` call in a function, while passing in the list comprehension as an argument, we will write a function such as `sum` as a macro.

An array comprehension – the application of the array function to a bounds and a list comprehension – is a special case of the application of `foldl` to a list comprehension. The initial value of the accumulator is an empty array of the appropriate bounds, and the accumulating function updates the array as it encounters each subscript/value pair.

Although elegant and expressive, array comprehensions may be difficult to implement efficiently on sequential machines. We have discussed how to eliminate the apparent proliferation of lists – but that turns out to be the least of our problems! In the next section we elaborate on the issues that must be faced.

4 Inefficiencies of Array Comprehensions

The metric against which we measure the efficiency of arrays is that provided by conventional imperative arrays—in particular, constant-time lookup, and constant-time/zero-space update. The two most obvious difficulties in achieving this goal with array comprehensions are: (1) the standard (but now more complicated) problem of overcoming the inefficiencies of lazy evaluation, and (2) the problem of avoiding the construction of the many intermediate lists that the second argument to `array` seems to need. But there is more to it than this, and thus we begin with a detailed discussion of the key sources of inefficiency:

Bounds checking. Avoiding bounds checking is equally difficult with functional and imperative arrays, and thus we ignore this issue in the present paper, since existing imperative language techniques work just as well. Indeed, many imperative languages eliminate this cost simply by turning off bounds checking altogether, at the expense of compromising program correctness. Ideally one would use properties of subscripts and loop indices to eliminate or at least lift bounds checks outside of loops.

Detecting write collisions. Only one "assignment" is allowed per element in an monolithic array; thus some mechanism must be provided to check for subscript/value pairs with the same subscript. In this paper we will describe a subscript analysis similar to detecting *output dependences* in imperative arrays, which can frequently establish at compile time that write collisions are impossible.

Detecting "empties". The bounds argument in an array comprehension establishes the size of the array, but there may not be an subscript/value pair for a particular subscript—the value at that position is thus undefined, and we call it an *empty*. Rather than check for the definedness of elements at run-time, we would like to know at compile-time that there are no empties. Such a condition exists if all of the following conditions hold:

- There are no write collisions.
- There are no out-of-bounds definitions.
- The number of subscript/value pairs is equal to the number of array elements.

If these conditions hold, the indices in the list of subscript/value pairs must be a permutation of the array's indices, therefore every subscript has a definition and there are no empties. In most cases, one can determine these conditions at compile time or, if that fails, in straight-line code before entering any loops.

The overhead of "thunks". The simplest, and in general the only, way to implement non-strict arrays is to represent each element as a "thunk" (i.e., a delayed representation of a value), which is evaluated upon demand. This is the standard way of realizing lazy evaluation, but the cost can be exorbitant in large arrays, at least in comparison to strict arrays. The cost includes the space and time overheads of creating, testing, and garbage collecting thunks. We will show how to use subscript analysis, similar to finding *true dependences* in imperative arrays, to find a safe schedule for evaluating elements without thunks.

Copying / trailers / reference counting. Expressing a new array as the compositions of an old array with new definitions for some elements creates the problem of deciding what to do with the old array – ideally we would like to re-use it; i.e. update it in place. In the case where no reference is made to the old array, this problem is the same as that faced in implementing incremental arrays, and has been studied extensively in the past [5, 11]. However in this paper we will discuss a variation of the problem—that in which the old array elements are used to create the new ones, and thus a safe schedule of the updates needs to be determined to do the updates in place. We will show that a subscript analysis, similar to detecting *anti-dependences* in imperative arrays, can frequently find a safe schedule for solving this problem.

Avoiding intermediate lists. This is perhaps the most obvious problem to solve, but it turns out to be rather easy! It amounts to a simple variation on the transformation of ordinary list comprehensions in [23], which is a special case of "deforestation" as proposed in [24]. All intermediate lists can be replaced by tail-recursive loops. Since this problem has been studied extensively elsewhere, we ignore it in this paper.

In conclusion, there are many inefficiencies that need to be overcome, and we will concentrate in this paper on detecting write collisions, eliminating thunks, updating arrays in place, and avoiding checks for the "definedness" of elements. The solutions to all of these problems involve some sort of subscript analysis. Indeed, recently there has emerged in the compiler community the folklore that the problem of compiling functional arrays efficiently for sequential execution has a lot in common with the problem of compiling imperative arrays efficiently for parallel machines. To some extent this is true, and not surprising once subscript analysis becomes necessary, but the details are non-trivial and the analogy is not as straightforward as one might think. One of the key contributions of our work is to make this relationship precise – to elucidate the transfer of the concepts of subscript analysis and *true, output,* and *anti-dependences* from the imperative to the functional framework.

5 Examples Of Dependence Graphs

The essential result of subscript analysis is a *dependence graph* that indicates which elements in an array depend on which others. It is similar to dependence graphs used in the imperative framework (for example, see [2, 7], and we give a few examples to demonstrate the idea.

An array expression will typically take a nested list comprehension for its subscript/value pair list. Each innermost intermediate list in this comprehension will be a singleton list specifying a subscript/value pair expression of the form `[ s := v ]`; we will call each such singleton list a *s/v clause*. A s/v clause plays a role very similar to an assignment statement in a DO loop specifying an array in an imperative language.

In a recursively defined array, we can draw a dependence edge between each *source* element that provides data and a corresponding *sink* element that needs that data. Each array element is computed as an *instance* of some s/v clause in the subscript/value pairs list comprehension. We can consider the dependence edge between a particular source element and sink element as an instance of a dependence edge between the corresponding source clause and sink clause.

When the subscript expressions are linear in the loop indices, and the loop bounds are statically known, compile-time subscript analysis allows us to construct a dependence graph with s/v clauses as vertices and dependence edges between s/v clauses. The analysis labels the edges with information about the *direction* of the dependence with respect to the loops surrounding the source and sink clauses; such labels are called `direction vectors` [2]. For example, an edge labeled (>) says that source and sink are surround by a single-level loop; for every instance of this edge, the source

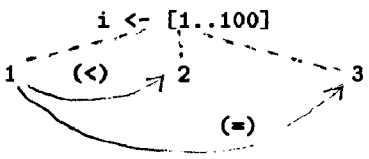
element is always computed at a "later" value of the loop index, relative to the "earlier" loop index value at which the corresponding *sink* element is computed.

We put quotation marks around "earlier" and "later" because we are talking about the relative position of a dependence's source and sink instances within the range of the loop index, *not* the temporal order in which those instances are executed at run time. However, once we know the direction-labeled dependence edges between s/v clauses of a list comprehension, we can use that information to decide whether the order specified by the list comprehension is safe for sequential compilation without thunks. The order is safe for thunkless compilation if for every edge in the dependence graph, the source instance is always computed before the sink instance. If the list comprehension's order is not safe, the direction-labeled dependence edges can frequently give us enough information to restructure the list comprehension to achieve a safe order, while preserving the semantics of the array expression.

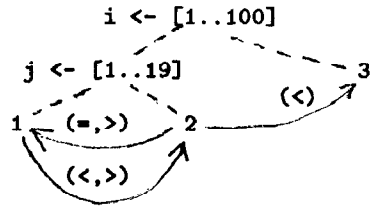
Let us represent a list comprehension as an expression tree; using the subscript analysis described in the next section, we will put labeled dependence edges between s/v clauses.

```
let a = array (1,300)
  [* [ 3*i   := ... ] ++    -- Clause 1
     [ 3*i-1 := ... a!(3*(i-1)) ... ] ++
        -- Clause 2
     [ 3*i-2 := ... a!(3*i) ... ]
        -- Clause 3

     | i <- [1..100] *]
```



In this example the labeled edge $1 \rightarrow 2$ $(<)$ says that clause 1 in an "earlier" instance of loop i supplies a value needed by clause 2 in a "later" instance of that loop. For thunkless sequential compilation, the loop must run temporally in the forward direction, from 1 to 100. The edge $1 \rightarrow 3$ $(=)$ says that within a single instance of loop i clause 1 supplies a value needed by clause 3. For thunkless compilation, clause 1 must be computed before clause 3 within a single instance, but the relative order of 1 and 2 or of 2 and 3 does not matter.



In this example edge $2 \rightarrow 1$ $(=,>)$ says that clause 2 supplies data to clause 1 in the same instance of the outer i loop, but from a "later" to an "earlier" instance of the inner j loop. The $2 \rightarrow 1$ $(=,>)$ edge requires that within a single i instance, the j loop must run in the backward direction (from 20 to 1) to permit thunkless compilation. The $1 \rightarrow 2$ $(<,>)$ and $2 \rightarrow 3$ $(<)$ edges agree that the i loop must run in the forward direction to permit thunkless compilation, but neither edge influences the inner j loop.

In the next section we explain how subscript analysis computes the labeled dependence edges between s/v clauses of the list comprehension expression tree. In the section on code generation we explain how to use this dependence information to generate a safe schedule for the list comprehension such that every dependence edge's source is computed before its sink, allowing compilation of a recursively defined array without thunks.

6 Subscript Analysis

We will briefly outline the adaptation of subscript analysis to functional monolithic arrays in the case of 1-dimensional subscripts. The analysis can be extended to multi-dimensional subscripts by ANDing tests on each dimension, or by linearization of the array; see [6]. Consider two references to array m, surrounded by d shared loops:

```
m = array bounds
  [*  ... m!(f i1 ... id) ...
      ... m!(g i1 ... id) ...
   | i1 <- [1..N1], ..., id <- [1..Nd] *]
```

We want to know if there is a dependence between two references to an array: do they refer to the same element?

Notice that the loops have been *normalized*: the low value of the index is 1, and the index increment is 1. When the subscript expressions are linear in the loop indices, the surrounding loops can always be put in normalized form ([21]). A huge number of scientific programs use only linear subscripts, and the derivation of subscript analysis algorithms assumes linear subscripts with normalized loops.

Definition: The *bounded integer solution test:* there exists a dependence between these two references to array `m` if and only if within the region of interest

$$R = \{(x_1, \ldots, x_d, y_1, \ldots, y_d) \mid x_1, y_1 \in [1..M_1], \ldots, x_d, y_d \in [1..M_d]\}$$

there exists an integer solution to the dependence equation

$$\begin{aligned} h \quad & x_1 \ldots x_d \, y_1 \ldots y_d \\ = \; & (f\ x_1 \ldots x_d) - (g\ y_1 \ldots y_d) \\ = \; & 0. \end{aligned}$$

□

In other words, there must be values for the loop indices that make the difference between the two subscripts equal zero.

Here x_k and y_k are two instances of the loop index i_k. But if a dependence exists, we may want further information: what is the direction of the dependence with respect to the surrounding loops? We get this information by further constraints on R; that is, by constraining the relative values of the loop indices for the two references. For example, the dependence edge label $(=,<,>,*)$ means a dependence

holds with the constraints $x_1 = y_1$, $x_2 < y_2$, $x_3 > y_3$, no constraint on the relative values of x_4 and y_4.

There are tractable decision algorithms when the subscript functions f and g are *linear* in the loop indices. Linear diophantine equation theory gives us an exact test (necessary and sufficient) for the existence of a bounded integer solution, but it is exponential in the number of surrounding loops ([25], [6]). For 1 or 2 levels of nesting this algorithm is reasonable. But from this definition of dependence we can derive two theorems give us inexact tests (necessary but not sufficient) that are linear in the depth of loop nesting.

Theorem 1: The *any integer solution test:* there is a dependence only if there is an integer solution to the dependence equation, regardless of whether the solution falls within R.

Theorem 2: The *bounded rational solution test:* there is a dependence only if there is a rational solution to the dependence equation within R.

The bounded integer test is an "if-and-only-if" definition of dependence. The two theorems weaken this definition to an "only-if", the first by dropping the bounds on the solution, the second by dropping the requirement that the solution be integer.

- *The GCD test* is derived from the first theorem. This test is the same for dependence testing in imperative and functional arrays. Let the linear subscripts be:

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

Label the d nested loops from the set $Q = [1..d]$, outermost first. Partition Q according to the constraints placed on x_k and y_k by region R:

- $Q_* =$ the set of all loops k with no constraints on x_k and y_k,
- $Q_< =$ the set of all loops k with the constraint $x_k < y_k$,

and so forth. There exists a dependence only if

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

- *The Banerjee inequality test* is derived from the second theorem. For linear subscripts, we can determine lower and upper bounds on the difference $(f\ x_1\ \ldots\ x_d) - (g\ y_1\ \ldots\ y_d)$ between the two subscripts. If those bounds do not bracket zero, then the two references cannot be to the same array element, and a dependence cannot exist. We will outline a variation on the proofs of Theorem 2 in [25] and Theorem 4 in [2] to find the these bounds using assumptions appropriate to functional arrays. Previous work assumes imperative array semantics, which depend upon the surrounding loops running in the specified direction, whereas functional array semantics are independent of loop direction.

We compute the bounds on the dependence equation expression by summing the bounds on its terms, one term for each surrounding loop. The kind of constraint put upon a term's loop indices determines how we compute its upper and lower bounds.

Definition: For integer t let us define the *positive part* t^+ and the *negative part* t^- as in [25] and [2]:

$$t^+ = \begin{cases} t & \text{if } t \geq 0, \\ 0 & \text{if } t < 0. \end{cases}$$

$$t^- = \begin{cases} -t & \text{if } t \leq 0, \\ 0 & \text{if } t > 0. \end{cases}$$

□

Lemma: Let $k \in Q_*$. We want to 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 – either instance of loop k can take on any value within the loop bounds $[1..M_k]$. The bounds on this term are:

$$\begin{aligned} &(a_k - b_k) - (a_k^- + b_k^+)(M_k - 1) \\ \leq\ &a_k x_k - b_k y_k \ \leq \\ &(a_k - b_k) + (a_k^+ + b_k^-)(M_k - 1) \end{aligned}$$

Proof: The bounds on the term $a_k x_k - b_k y_k$ equals the sum of the bounds on the terms $a_k x_k$ and $-b_k y_k$. Let us find the bounds on $a_k x_k$. Let $x_k = 1 + s_k$. Then

$$\begin{aligned} &a_k x_k = a_k + a_k s_k, \\ &0 \leq s_k \leq M_k - 1. \end{aligned}$$

Then by Lemma 1 in [2]:

$$\begin{array}{lll} -a_k^-(M_k - 1) & \leq a_k s_k & \leq a_k^+(M_k - 1), \\ a_k - a_k^-(M_k - 1) & \leq a_k x_k & \leq a_k + a_k^+(M_k - 1). \end{array}$$

Similar reasoning give us bounds:

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

Adding the bounds on terms $a_k x_k$ and $-b_k y_k$ give us the bounds on $a_k x_k - b_k y_k$. □

Theorem: Given a partition $Q = Q_= \cup Q_< \cup Q_> \cup Q_*$ placing a constrained region of interest R on the shared loops surrounding two subscript expressions $(f\ x_1\ \ldots\ x_d)$ and $(g\ y_1\ \ldots\ y_d)$, the difference $(h\ x_1\ \ldots\ x_d\ y_1\ \ldots\ y_d) = (f\ x_1\ \ldots\ x_d) - (g\ y_1\ \ldots\ y_d)$ between these subscripts can equal zero *only if* the following expressions for h's minimum and maximum bracket zero:

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

Proof: We developed the lower and upper bounds on a term for any loop $k \in Q_*$. By similar proofs, minima and maxima can be derived for $k \in Q_<$, $Q_=$, and $Q_>$ in which region R constrains the relative values of x_k and y_k (see [2],

[25]). Gathering these inequalities we get the lower and upper bounds on the difference between the two subscripts. The difference between the two subscripts can only be zero if the bounds on the difference brackets zero. □

This inequality must be satisfied for a dependence to exist, given this set of constraints. Of course, since this is a necessary but not sufficient test, the satisfaction of this inequality does not tell us that a dependence does exist, but only that we cannot prove that it does not exist *given these constraints*. Naturally we start with no constraints, e.g., $(*,*,*)$ for a 3-level nested loop. If a dependence is impossible with no constraints, then certainly it is impossible with sharper constraints.

Given loop nesting depth n, a single exact test costs $O(c^n)$ time. The GCD test costs $O(n)$ time, but if it is satisfied we only know that a dependence is possible, and we know nothing about the direction constraints on the possible dependence. The Banerjee test costs $O(n)$ time, and when constraints are given on R it tells us gives a direction vector label for the possible dependence. Unfortunately, we need $O(c^n)$ different Banerjee tests in the worst case to completely determine the direction vector on a dependence. However, [6] suggests a search tree approach to refining the constraints on the region R for the Banerjee test. In many cases the search tree approach gives complete information on any possible dependence between a pair of array references in $O(n)$ or even $O(1)$ time.

So far we have presented the Banerjee test for shared loops. If either array reference is further surrounded by unshared loops, we need to compute the unshared loops' contributions to the bounds.

Lemma: If loop k over $[1..M_k]$ surrounds subscript $(f\ x_1...)$, but does not surround the other subscript $(g\ y_1...)$, then the contribution the loop k term makes to the bounds on h is:

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

Likewise, if loop k over $[1..N_k]$ surrounds $(g\ y_1...)$ but not $(f\ x_1...)$, then the contribution the loop k term makes to the bounds on h is:

$$-b_k - b_k^+(N_k - 1) \leq -b_k y_k \leq -b_k + b_k^-(N_k - 1). \ \square$$

Proof: Simply use the proof for unconstrained x_k and y_k, but drop the contribution of the subscript that is not contained within loop k.

7 Detecting Write Collisions

Only one "assignment" is allowed per element in an monolithic array; thus some mechanism must be provided to check for subscript/value pairs with the same subscript. Detecting write collisions at compile time is similar to the problem of detecting *output dependences* in imperative arrays: do two instances of s/v clauses (in an imperative language, two instances of assignments statements) write to the same element?

The GCD and Banerjee tests are necessary but not sufficient: they can show that a write collision *cannot* occur, but they cannot show that a collision definitely will occur, only that it may potentially occur.

If subscript analysis shows us that no two s/v clause instances can write to the same element, we do not compile any runtime code to check for collisions. If an inexact subscript test says a collisions is possible, we must compile collision testing code, and inform the programmer of the risk. If an exact subscript test says a collision will definitely happen, we flag an error.

For an accumulated array, which allows multiple values to be combined into a single element, the combining function may be non-commutative. If this is the case, any rescheduling of the order of evaluating s/v pairs must maintain the order in which values are combined into an element. Write collisions edges then become true output dependence edges, and ordering information on these edges put a constraint on the permissible scheduling.

8 Thunkless Code Generation

Our next goal is to compile an array so there is no need for thunks: for every dependence edge between two elements of the array, we want to make sure the source is always computed before the sink. The compiler needs to be concerned with two kinds of dependence edges [2]:

- *Loop-carried dependences* in which the sink and source occur in different loop instances, such as $(<)$ (source is in an "earlier" instance than sink), $(>)$ (source is in a "later" instance than sink), $(=,<)$ (source is in the same outer loop instance as sink, but an "earlier" inner loop instance);
- *Loop-independent dependences* in which the sink and source occur in the same loop instance, such as $(=)$ or $(=,=)$, or do not share a loop at all, such as $()$.

We will develop the principles for single-level loops, then generalize to nested loops.

8.1 Thunkless Code For a Single-Level Loop

In a single-level loop, the loop-carried dependence $(<)$ and $(>)$ determine the direction of the loop, whereas the loop-independendent dependences $(=)$ determine the order in which s/v clauses are computed within a single loop instance. We can ignore $(=)$ edges when determining the loop direction (unless we choose to split the loop into multiple passes; see below), and we can ignore $(<)$ and $(>)$ edges when determining the order of clauses within a single loop instance.

8.1.1 Loop Direction Scheduling

To determine the loop direction we have these cases:

- There are no loop-carried edges: all edges are $(=)$ with sink and source in the same instance. The loop may run either direction. The graph may be either cyclic or acyclic.

- The loop-carried edges are all (<) (or all (>)). Then simply choose the loop direction that guarantees computation of sources before sinks. The graph may be either cyclic or acyclic.
- Both (<) and (>) edges are present. This case is more difficult, and is the subject of the next section.

8.1.2 Loop Direction Scheduling With Both (<) and (>) Edges

When an array expression contains both (<) and (>) edges, our algorithm for scheduling loop direction depends on whether the dependence graph:

- acyclic,
- cyclic with no cycle containing both (<) and (>) edges, or
- cyclic with at least one cycle containing both (<) and (>) edges.

A dependence graph is cyclic if at least one of its SCCs (strongly connected components) contains more than a single vertex. If a SCC contains both (<) and (>) edges, then that SCC must contain a cycle with both edge types: since every vertex in an SCC can reach every other vertex in that SCC, every pair of (<) and (>) edges in an SCC must take part in a cycle. Computing the SCCs of a graph $G = (V, E)$ costs $O(\max(|V|, |E|))$ time, and we inspect all $|E|$ edges to classify the graph's SCCs.

• *Acyclic graph with both (<) and (>) edges.* Wrap each vertex's corresponding s/v clause in a separate loop, and schedule the loops consecutively as ordered by topological sort. The sort must consider *both* loop-carried and loop-independent (=) edges.

For example, consider the graph

$$V = \{A, B, C\},$$
$$E = \{A \rightarrow B\ (<),\ B \rightarrow C(>),\ A \rightarrow C(=)\}.$$

Since A depends on no other clauses, we can compute it in a loop running *either* direction. When the A loop is complete, we can compute B in a separate loop also running *either* direction; although B depends on A from an "earlier" loop instance, splitting the loop into two loops which compute all As before all Bs also satisfies the dependence, no matter which way the two loops run. Likewise, schedule the C loop after the A and B loops.

This approach – a separate loop pass for each s/v clause – works for any acyclic graph, but when several nodes are connected by dependence edges agree that on a loop direction, we can collapse the separate passes for these nodes into a single loop. In the example above there are 3 different schedules that can collapse the 3 loops into 2 loops, with less loop overhead. A static scheduling algorithm for acyclic dependence graphs is given in the next subsection.

• *Cyclic graph with both (<) and (>) edges, but no cycle contains both edge types.* Consider the quotient graph we get by collapsing each SCC to a single vertex, and discarding all edges internal to a SCC. This quotient graph is a DAG which can be scheduled by a slight modification of the scheduling algorithm for the acyclic graph described above. If an SCC is a single vertex or contains only (=) edges, its surrounding loop may run either direction; if it contains any (<) edges ((>) edges) it must run in the forward (backward) direction.

• *Cyclic graph with at least one cycle containing both (<) and (>) edges.* We cannot schedule the loop in a way that guarantees every source is computed before every sink. Consider the graph $V = \{A, B\}, E = \{A \rightarrow B\ (<),\ B \rightarrow A(>)\}$. We cannot run a single loop, either forward or backward, that satisfies both dependence edges, nor can we split the loop into two loops. We have no choice but to compile using thunks. However, monolithic array comprehensions involve only flow dependences; for loops updating incremental arrays, a dependence cycle involving an antidependence edge can be broken.

8.1.3 Scheduling an Acyclic Dependence Graph With Both (<) and (>) Edges

Static scheduling algorithm: Suppose we want to schedule the first pass in the forward direction, which satisfies (=) and (<) but not (>) dependences.

Definition: A node must be marked as 'not-ready' for a forward direction loop pass if it is reachable from any root (a node of in-degree zero) in the DAG via any path that includes at least one (>) edge. Otherwise a node is marked 'ready.' □

Schedule all the 'ready' nodes in the first pass, then delete them from the graph. Repeat until there are no nodes left. At every step the graph is a DAG, every DAG contains at least one root, and every root is 'ready', so at least one node gets deleted each step, and the scheduling algorithm terminates in at most $|V|$ steps.

Obviously the most common dependence graphs open room for many refinements. Often all the loop-carried dependence edges leaving the roots will specify the same dependence direction, so it makes sense to schedule the first pass in a direction consistent with that dependence direction.

Algorithm to detect 'not-ready' nodes: The algorithm is a modified depth first search over a DAG. As we visit each node i, maintain a variable s describing the path from the current spanning tree root r to node i:

- s = 'ready' if the path contains no (>) edges
- s = 'not-ready' if the path contains at least one (>) edge.

Each node i has two variables: $i.visited$ and $i.ready$. As usual in DFS, $i.visited$ = false initially. The initial value of $i.ready$ does not matter, but let it be 'ready'. When we visit a node:

- If $i.visited$ = false, then set $i.visited$ = true, set $i.ready$ = s, and continue to visit i's children.
- If $i.visited$ = true, and $i.ready$ = either 'ready' or 'not-ready', and s = 'ready', then backtrack.

- If $i.visited$ = true, and $i.ready$ = 'not-ready', and s = 'not-ready', then backtrack.
- If $i.visited$ = true, and $i.ready$ = 'ready', and s = 'not-ready', then $i.ready = s$ = 'not-ready', and continue to revisit i's children.

The algorithm behaves exactly like DFS except in the last case, in which a node has already been visited by a 'ready' path (no (>) edges), and then is revisited via a 'not-ready' path (at least one (>) edge). If an already visited node gets remarked from 'ready' to 'not-ready', then all its'ready' descendants must also be remarked 'not-ready'.

In the worst case this algorithm will visit every node twice and cross every edge twice. The worst case time complexity of this algorithm is therefore the same as DFS, $O(\max(|V|, |E|))$.

8.1.4 Scheduling Within a Single Instance of a Loop

Drop (<) and (>) edges (which can only influence scheduling across loop instances) from the graph and consider only (=) edges.

- If the (=) edge graph is acyclic, schedule using topological sort.
- If the (=) edge graph is cyclic, then we cannot choose a safe thunkless schedule: compile using thunks.

8.2 Scheduling Thunkless Code For Nested Loops

Let us start at the outermost loop. How do we decide loop direction? How do we decide the ordering of s/v clauses and nested loops within a single instance of the outermost loop? For now we will not pursue any more drastic restructurings, such as interchange of loop nesting levels.

Let us collapse each nested inner loop into a single *entity*, retaining edges between s/v clauses of the inner loop as self-cyclic edges in the new dependence graph. We treat the outer loop as a single-level loop containing a set of entities (s/v clauses and inner loops) with no internal structure. Let us consider the role of each kind of dependence edge.

8.2.1 Scheduling the Outermost Loop

(<), (>), and (=) dependence edges connect source and sink array references that only involve the outermost loop. Such edges can therefore be treated exactly as they are for single-level loops. A (<) edge requires the outer loop to run forward for the source entity to precede the sink entity. A (>) edge requires a backward loop. A (=) edge requires the source entity to precede the sink entity within a single outer loop instance.

(<, ..), (>, ...), or (=, ...) dependence edges connect source and sink array references that involve an inner loop as well. Since the inner loop (or loops) involved in such a dependence must be shared by both source and sink, such edges look as if they are self-cyclic upon a single entity from the outermost loop's point of view.

However, notice that a (<, ..) or (>, ...) edge can be treated exactly like a (<) or (>) edge whose source and sink are the same s/v clause: it influences outer loop direction, but has no effect on entity ordering within a single instance.

A (=, ...) dependence edge, on the other hand, can be ignored at the level of the outer loop. Since its source and sink occur within a single instance of the outer loop, it is not loop-carried with respect to the outer loop, and cannot influence the outer loop direction. And since its source and sink are within a single entity (the same inner loop) from the outer loop's point of view, a (=, ...) edge cannot influence entity ordering at the level of a single instance of the outer loop.

8.2.2 Scheduling the Inner Loops

We have scheduled the direction of the outer loop and ordering of its entities. When we generate code for one of the nested inner loops, it is easy to see which nodes belong in the inner loop's dependence subgraph – they are simply the s/v clauses appearing in that inner loop – but which edges do we include? We exclude:

- all edges to or from nodes excluded from the inner loop;
- all (<), (>), and (=) edges, since they only influence scheduling at the level of the outermost loop;
- all (<, ...) and (>, ...) edges, since they connect source and sink in separate outer loop instances, and we are generating code for the inner loop within a single instance of the outer loop.

We keep only (=, ...) edges: they are the only edges relevant to generating code for an inner loop.

For example, a (<, =) or (>, <) edge influences loop direction for the outer loop, but has no influence whatever on the inner loop. A (=, <) or (=, =) edge has no influence whatever on the outer loop, but (=, <) influences inner loop direction, and (=, =) influences entity ordering in a single instance of the inner loop. In the terminology of [2], (>, <) is loop-carried at level 0 (at the outer loop), and (=, <) is loop-carried at level 1 (at the inner loop).

8.2.3 Summary of Scheduling For Nested Loops

In summary, as we generate code for an outer loop, we treat each inner loop as a single entity. We treat (<, ...) and (>, ...) edges as if they were (<) and (>) edges for scheduling outer loop direction. We consider (=) edges, but ignore (=, ...) edges for entity ordering within a single outer loop instance.

When we recursively generate code for one of the inner loops, we drop all but (=, ...) edges. We can strip off the leading = and treat this inner loop as if it were now the outermost loop.

9 Making Updates Single-Threaded

Often a new array is most conveniently expressed as the composition of an old array with new definitions for some section of the array. In this section we define bigupd, a useful construct for specifying updates in a semi-monolithic manner; that is, over a large piece of an array. We show how subscript analysis can let us convert simple incremental programs that apparently prohibit in-place update into more complicated programs that permit in-place update with minimal or no copying.

Although monolithic and incremental arrays are of equal semantic power – either can be expressed in terms of the other – some algorithms are more clearly expressed monolithically, some incrementally. The incremental approach is more expressive (1) when the result array has nearly the same contents as the input array, with only a few elements changed, or (2) when the result array completely changes the input array, but the result can overwrite the input in place, at a great savings in storage.

Updating an array in place is only possible if there are no other references to the old version of the array – the old version is no longer live data.

Definition: If at every update to an array, there are no references to the old version of the array, we say that the array is *single-threaded.*

Single-threadedness is an operational rather than semantic concept. A variety of run-time schemes, such as reference counts, as well as compile-time schemes, such as abstract reference counting and path analysis, have been proposed to make sure updates are single-threaded [5, 12]. Array trailers are a run-time scheme that assumes an update is not single-threaded, but gives reasonably good performance when it is single-threaded. So far, none of these schemes have been cheap enough to use in a practical compiler. However, much promise is shown by [8], in which a polymorphic type system is guarantee that an incremental array is updated only in a single-threaded manner.

All these schemes require the programmer to make sure the array is referenced in a single-threaded manner. But for many problems the most mathematically expressive form is to write new element values by refering to the original array, rather than to an intermediate version of the array. Unfortunately, such a form is not single-threaded in its use of the array, preventing in-place update.

But notice that single-threadedness is too strong a requirement for in-place update. We do not require that the entire old version of the array be dead, only that the element to be updated be dead. If we can determine at compile time that an update changes a dead element, then any remaining references to the old version of the array refer to elements that are unchanged in the new version. We can make the update single-threaded by changing the remaining old version references into new version references.

In many common cases, the dependence information from subscript analysis can be used to transform a simple, non-single-threaded form, which may require extensive run-time copying, into a more complicated, single-threaded form that requires no run-time copying, or only as much run-time copying as the equivalent hand-coded form. Let us define

```
bigupd a svpairs = foldl upd a svpairs
```

As we discussed earlier, we can treat `foldl` as syntax such that when `svpairs` is written as a list comprehension over arithmetic sequence generators, the entire expression can be compiled as tail-recursive DO loops. Consider this code fragment from LINPACK for swapping rows i and k of a matrix.

```
let* a' = bigupd a
    [* [ (i,j) := a!(k,j) ] ++
       [ (k,j) := a!(i,j) ]
       | j <- [1..n] *]
```

Assume that at the time bigupd is applied, a has no other references – a is single-threaded w.r.t bigupd. The two s/v clauses are involved in an antidependence cycle, each edge of which is labeled (=). A cycle including at least one antidependence edge can always be broken by node-splitting:

```
let* a' = bigupd a
    [* let temp = a!(i,j) in
       ( [ (i,j) := a!(k,j) ] ++
         [ (k,j) := temp ] )
       | j <- [1..n] *]
```

If there are no antidependence cycles, the loops can always be scheduled so that no live value is overwritten. Simply treat the antidependence edges as dependence edges, and schedule using the techniques in the thunkless compilation section.

We have determined a schedule of selects and updates such that whenever an update takes place, the array as a whole may still be live, but the element to be updated is dead. Since every select from the original array is scheduled before an update kills that element, therefore we can convert all selects from the original array to selects from the current intermediate version of the updated array. If the array is single-threaded entering bigupd, it is now guaranteed to remain single-threaded within bigupd.

Antidependence cycles can always be broken by node-splitting, but we want to do better than copying the entire array. Consider this simplified version of a single step of Jacobi iteration:

```
let* u' = bigupd u
    [* (i,j) := u!(i,j-1) + u!(i,j+1) +
                u!(i-1,j) + u!(i+1,j)
       | i <- [1..m]; j <- [1..n] *]
```

Each reference to the old matrix gives this clause a self-cyclic antidependence edge. The inner j loop carries antidependence edges in both directions, $(=,<)$ and $(=,>)$. If there were only one such cycle (say the $(=,<)$ edge) we could ensure that a value is not overwritten before it is used – satisfy the antidependence – by scheduling the loop in the right direction.

Node-splitting can eliminate the conflicting self-cycle (the $(=,>)$ edge) by writing the value into a temporary to be carried over to the next j loop iteration. But notice that the distance of the loop-carried anti-dependence must be constant w.r.t. the loop. If instance j depends on data overwritten in instance $j-k$, we must keep k temporaries. Usually the distance will be 1.

Likewise, node-splitting can eliminate one of the outer i loop antidependence cycles, but even when the dependence distance is 1, the temporary must be a vector large enough to hold all the live values that may be overwritten by the inner loop. Nevertheless, node-splitting requires a factor n fewer copies than naive compilation, where the outer loop has n instances.

Our examples so far required some copying, but it is much more common that a code can be converted to single-threaded form with no copying. Examples from LINPACK include scaling a matrix row, and in-place SAXPY. Here is a version of a single step of Gauss-Seidel or SOR iterative solution of an elliptic PDE on a 2D mesh, simplified to make the dependences clear. Livermore Loops Kernel 23 (2-D implicit hydrodynamics code fragment) has the same northwest-to-southeast wavefront structure.

```
u' = bigupd u
                    δ̄(<,=)
          δ̄(=,<)
[* (i,j) := u!(i,j+1) + u!(i+1,j) +
            u'!(i,j-1) + u'!(i-1,j)
          δ(=,<)        δ(<,=)

 | i <- [2..m-1]; j <- [2..n-1] *]
```

The dependence graph has a single node with four self-cyclic edges: true dependence edges $\delta(<,=)$ and $\delta(=,<)$ and antidependence edges $\overline{\delta}(<,=)$ and $\overline{\delta}(=,<)$. At both loop levels the direction of the loop-carried dependences agree. By scheduling the loop directions appropriately, the true dependences can be satisfied without compiling thunks, and the antidependences without copying.

In conclusion, antidependence edges can be treated exactly like true dependence edges for the sake of static scheduling. The goals are different – the schedule must satisfy all dependences to avoid compiling thunks, and must satisfy all antidependences to avoid unnecessary copying – but the scheduling algorithms presented in the previous section do not need to distinguish true dependence edges from antidependence edges. Certain kinds of dependence cycles – with all $(=)$ edges or with both $(<)$ and $(>)$ edges – prevent static scheduling, but when at least one edge in the cycle is an antidependence, it can be broken by node-splitting. Node-splitting usually requires much less copying than naive compilation; in the examples we have tried, it requires exactly as much copying as a hand-coded program.

10 Further Research

Subscript analysis was originally developed within the imperative language community to permit vectorization and parallelization of sequential programs. We have adapted this analysis to the efficient sequential execution of functional language programs. But obviously this analysis can also be extended to the vectorization and parallelization of functional language programs as well. As with imperative languages, such transformations on functional language programs needs to focus on finding innermost loops with no loop-carried dependences.

Vectorization requires the elements of the argument vectors and the result vector be treated as floating point data, so we need to know the arrays are used in a strict context. Vectorization techniques are likely to be important not only on expensive architectures like the Cray, but in the coming years on workstations based on processors such as the Intel i860 and IBM RIOS. These RISC-like processors have on-chip floating point hardware that can execute a floating point instruction every clock cycle, if the compiler can achieve such as schedule.

Efficient use of the memory hierarchy has a tremendous impact on performance in scientific computing. [1] applies the dependence information from subscript analysis to the problem of enhancing the paging performance of imperative language programs through program transformation. Such transformation are especially important for functional languages, since their focus on 'what' is computed rather than 'how' gives less control over such operational behavior.

11 Conclusions

Although non-strict monolithic arrays are not perfect solutions for all applications of arrays, they are very expressive in the majority of application areas, and many interesting applications have already been programmed using them (see [13] for more examples). What we have demonstrated in this paper is that they also can be compiled efficiently for sequential machines; in most applications we can remove the main sources of inefficiency that would otherwise prevent performance comparable to Fortran.

References

[1] D.J.Kuck Abu-Sufah, W. and D.H.Lawrie. On the performance enhancement of paging systems through program analysis and transformation. *IEEE Trans. Computers*, C-30(5):341–355, 1981.

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

[3] S. Anderson. Non-strict arrays in a strict context. Research report, Yale University, Department of Computer Science, January.

[4] V. Balasundaram. *Interactive Parallelization of Numerical Scientific Programs.* PhD thesis, Computer Science Department, Rice University, 1989.

[5] A. Bloss. Update analysis and efficient compilation of functional aggregates. In *FPCA*, pages 26–38, 1989.

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

[7] D.A.Padua B.Leasure D.J. Kuck, R.H.Kuhn and M.J.Wolfe. Dependence graphs and compiler optimization. In *ACM POPL*, pages 207–217, 1981.

[8] J. Guzman and P.Hudak. Single-threaded polymorphic lambda calculus. In *ACM Conf. on Logic in Computer Science*, 1990.

[9] P. Hudak. Alfl reference manual and programmer's guide. Research Report YALEU/DCS/RR-322, Second Edition, Yale University, October 1984.

[10] P. Hudak. Arrays, non-determinism, side-effects, and parallelism: A functional perspective. In *Proceedings of the Santa Fe Graph Reduction Workshop*, pages 312–327. Los Alamos/MCC, Springer-Verlag LNCS 279, October 1986.

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

[12] P. Hudak. A semantic model of reference counting and its abstraction (detailed summary). In *Proceedings 1986 ACM Conference on LISP and Functional Programming*, pages 351–363. ACM, August 1986.

[13] P. Hudak and S. Anderson. Haskell solutions to the language session problems at the 1988 Salishan high-speed computing conference. Technical Report YALEU/DCS/RR-627, Yale University, Department of Computer Science, January 1988.

[14] P. Hudak and P. Wadler (editors). Report on the functional programming language haskell. Technical Report YALEU/DCS/RR666, Yale University, Department of Computer Science, November 1988.

[15] P. Hudak and P. Wadler (editors). Report on the programming language haskell, a non-strict purely functional language (version 1.0), October 1989. to appear in SIGPLAN Notices.

[16] K. Iverson. *A Programming Language.* Wiley, New York, 1962.

[17] R.M. Keller. Fel programmer's guide. AMPS TR 7, University of Utah, March 1982.

[18] J.R. Mcgraw. The VAL language: description and analysis. *TOPLAS*, 4(1):44–82, January 1982.

[19] R.S. Nikhil, K. Pingali, and Arvind. Id Nouveau. Computation Structures Group Memo 265, Massachusetts Institute of Technology, Laboratory for Computer Science, July 1986.

[20] Jr. Steele, Guy L. and W. Daniel Hillis. Connection machine lisp: Fine-grained parallel symbolic processing. In *Proceedings 1986 ACM Conference on Lisp and Functional Programming*, pages 279–297, Cambridge, Massachusetts, August 1986. ACM SIGPLAN/SIGACT/SIGART.

[21] 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.

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

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

[24] P. Wadler. Deforestation. In *LNCS 300: European Symposium on Programming.* Springer-Verlag, 1988.

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