I-Structures: Data Structures for Parallel Computing

Arvind*
Rishiyur S. Nikhil
Keshav K. Pingali†

TR 87-810
February 1987

Department of Computer Science
Cornell University
Ithaca, New York 14853-7501

*This research was done at the MIT Laboratory for Computer Science. Funding for this project is provided in part by the Advanced Research Projects Agency of the Department of Defense under the Office of Naval research contract N00014-84- K-0099.

†Keshav Pingali was also supported by an IBM Faculty Development Award.

I-Structures: Data Structures for Parallel Computing

Arvind† (MIT)
Rishiyur S. Nikhil† (MIT)
Keshav K. Pingali‡ (Cornell University)

Abstract

It is difficult simultaneously to achieve elegance, efficiency and parallelism in functional programs that manipulate large data structures. We demonstrate this through careful analysis of program examples using three common functional data-structuring approaches— lists using `Cons` and arrays using `Update` (both fine-grained operators), and arrays using `make-array` (a "bulk" operator). We then present I-structures as an alternative, defining precisely the parallel operational semantics of Id, a language with I-structures. We show elegant, efficient and parallel solutions for the program examples in Id. I-structures make the language non-functional, but do not raise determinacy issues. Finally, we show that even in the context of purely functional languages, I-structures are invaluable for implementing functional data abstractions.

1 Introduction

There is widespread agreement that only parallelism can bring about significant improvements in computing speed (several orders of magnitude faster than today's supercomputers). Unfortunately, most of the computing models that we are comfortable and familiar with are based on current von Neumann, sequential architectures, and do not seem extensible in any straightforward way to parallel machines.

Functional languages have received much attention as appropriate vehicles for programming parallel machines, for several reasons. They are high-level, "declarative" languages, insulating the programmer from architectural details. Their operational semantics in terms of rewrite rules offers plenty of exploitable parallelism, freeing the programmer from having

[0] *Authors' addresses:*
† MIT Lab. for Computer Science, 545 Technology Square, Cambridge, MA 02139, USA
‡ Dept. of Computer Science, 303A Upson Hall, Cornell University, Ithaca, NY 14850, USA

This research was done at the MIT Laboratory for Computer Science. Funding for this project is provided in part by the Advanced Research Projects Agency of the Department of Defense under the Office of Naval Research contract N00014-84-K-0099.

Keshav Pingali was also supported by an IBM Faculty Development Award.

This paper will also appear in the proceedings of the Los Alamos Workshop on Graph Reduction(October 1986) and as M.I.T. CSG Report 269.

to "identify" parallelism explicitly. They are *determinate*, freeing the programmer from details of scheduling and synchronization of parallel activities.

In this paper, we focus on the issue of *data structures*. We first demonstrate some difficulties in the treatment of data structures in functional languages, and then propose an alternative, called "I-structures". Our method will be to take some test applications, and compare their solutions using functional data structures, and using I-structures. We study the solutions from the point of view of

- efficiency (amount of unnecessary copying, speed of access, number of reads and writes, overheads in construction, *etc.*),
- parallelism (amount of unnecessary sequentialization), and
- ease of coding.

We hope to show that it is very difficult to achieve all three objectives using functional data structures.

Since our ideas about I-structures evolved in the context of scientific computing, most of the discussion will be couched in terms of *arrays*.[1] All our program examples are written in Id, which is a functional language augmented with I-structures. It is the language we use in our research on parallel architectures. Of course, the efficiency and parallelism of a program depend on the underlying implementation model. Our findings are based on our own extensive experience with dataflow architectures— in particular the MIT Tagged-Token Dataflow Architecture, the centerpiece of our research [3,15]. We have also carefully studied other published implementations of functional languages. However, it is beyond the scope of this paper to delve into such levels of implementation detail, and so we conduct our analyses at a level which does not require any knowledge of dataflow on the part of the reader. In Section 5, we present an abbreviated version of the rewrite-rule semantics of Id, which captures precisely the parallelism of the dataflow machine; we leave it to the intuition of the reader to follow the purely functional examples prior to that section.

While the addition of I-structures takes us beyond functional languages, Id does not lose any of the properties that make functional languages attractive for parallel machines. In particular, Id remains a higher-order, determinate language, *i.e.*, its rewrite-rule semantics remains confluent. In the final section of the paper, we discuss the implications of such an extension to a functional language. We also show that I-structures are not enough— there are some applications that are not solved efficiently whether we use functional data structures or I-structures. This class of applications is a subject of current research.

2 The Test Problems

In this section we describe four small example applications which we use to study functional data structures and I-structures.

[1]However, it would be erroneous to infer that our conclusions are relevant only to programs with arrays.

2.1 Example A

Build a matrix with

$$A[i,j] = i + j$$

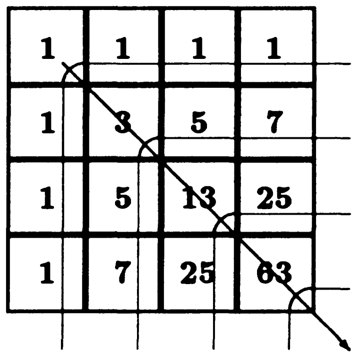
Note that the computation for each element is independent of all the others.

2.2 Example B (Wavefront)

Build a matrix with:

$$A[1,j] = 1$$
$$A[i,1] = 1$$
$$A[i,j] = A[i-1,j] + A[i-1,j-1] + A[i,j-1]$$

The left and top edges of the matrix are all 1. The computation of each remaining element depends on its neighbors to the left and above. In a parallel implementation one can thus imagine the computation proceeding as a "wavefront" from the top and left edges to the bottom-right corner of the matrix:



2.3 Example C (Inverse Permutation)

This problem was posed to one of us (Arvind) by Barendregt. Given a vector B of size n containing a permutation of integers $1..n$, build a new vector A of size n such that:

$$A[B[i]] = i$$

The computation for each of A's components is independent of the others. (This is called an inverse permutation because the result A also contains a permutation of $1..n$, and when the operation is repeated with A as argument, the original permutation is returned.)

2.4 Example D (Shared Computation)

Build two arrays A and B of size n such that

$$A[i] = f\ (h\ i)$$
$$B[i] = g\ (h\ i)$$

such that the h part of the computation for every i'th element of the two arrays is shared.[2]

This example illustrates shared computation across arrays. Sharing could also occur across indices in a single array— for example, the computations for $A[2i]$ and $A[2i+1]$ may have a common sub-computation. And of course, in other applications the two types of sharing may be combined.

3 Fine-Grained Functional Data Structure Operations

We begin by looking at two data-structuring operations traditionally found in functional languages. In Section 3.1, we look at "`Cons`", a pairing operation, and in Section 3.2, we look at "`Update`", an operation that specifies a single, incremental change in an array. We call them "fine-grained" operations because more useful operations such as a vector sum, matrix multiplication, *etc.* must be programmed in terms of a number of uses of these primitives.

3.1 `Cons`: Simulating Large Data Structures Using Lists

Functional languages have traditionally had a two-place "`Cons`" constructor as a basic data-structuring mechanism. Given `Cons`, one can of course write suitable array abstractions as a first step towards solving our examples. In this section we quickly reject this as a serious solution.

A typical representation for arrays using `Cons` would be to maintain an array as a list of elements, a matrix as a list of arrays, and so on. An abstraction for a general access to an array component may be defined as follows:

```
Def select A i = If (i == 0) Then hd A
                 Else select (tl A) (i-1) ;
```

Because of the list traversal, selection takes $O(n)$ reads, where n is the length of the array.

Now consider a vector sum, programmed in terms of `select`:

```
Def vector_sum A B i = If (i > n) Then nil
                       Else cons ((select A i) + (select B i))
                                 vector_sum A B (i+1) ;
```

[2]Here we use juxtaposition to indicate function application— notation that is common in functional languages. Application associates to the left, so that "`f x y`" stands for "`(f x) y`".

This function performs $O(n^2)$ reads, where a corresponding FORTRAN program would perform only $O(n)$ reads.

This problem can be mitigated at the expense of ignoring the **select** abstraction and taking advantage of the underlying list representation so that the list-traversing overhead is not cumulative:

```
Def vector_sum_2 A B = If (null A) Then nil
                       Else cons ((hd A) + (hd B))
                                 vector_sum_2 (tl A) (tl B) ;
```

This solution performs $O(n)$ reads (though it is still inefficient because it is not tail-recursive).

Unfortunately, *every* new abstraction must be carefully recoded like this because combinations of given abstractions are not efficient. For example,

```
vector_sum_2 A (vector-sum_2 B C)
```

creates and traverses an intermediate list unnecessarily.

Coding new abstractions efficiently is difficult because the list representation dictates a preferred order in which arrays should be constructed and traversed, an order that is extremely difficult to circumvent. Consider one of the most basic array operations: multiplication of two matrices **A** and **B** as described in a mathematics textbook:

$$C[i,j] = A[i,*] \circ B[*,j]$$

where $\circ$ is the "inner-product". But this requires a traversal of **B** by column, which is very inefficient in our list representation. One may propose first to transpose **B**; but even a transpose is not easy to code efficiently (we invite the reader to attempt it!), and even if it were, we still pay the overhead of making an intermediate copy of the matrix.

Finally, the use of a "fine-grained" data-structuring primitive such as **Cons** places an enormous burden on the storage allocator because of the large number and frequency of requests. Note also that in many typical implementations where a **Cons** cell occupies twice the storage of a number (for two pointers), the storage requirements for the list representation of a vector of numbers can be more than twice the storage for the numbers alone.

For the rest of the paper, we will assume primitives that allocate contiguous storage for each array, so that there is not much storage overhead, and so that array accesses take constant time.

3.2 Update: A Functional Array Operator

Instead of simulating arrays using lists, one could provide array operators directly. We now describe one such set of operators.

An array is allocated initially using the expression

```
array (l,u)
```

which returns an array whose index bounds are `(l,u)`, and all of whose locations contain some standard initial value (call it "`nil`").[3]

The expression

```
update A i v
```

returns an array that is identical to "`A`", except at index "`i`", where it contains the value "`v`". Despite its imperative-sounding name, this is a *functional* operation— it returns a *new* array and does not disturb `A`.

A component of an array is selected using the expression

```
A[i]
```

which returns the value at index "`i`" from array "`A`".

For multi-dimensional arrays, we could nest 1-dimensional arrays, or we could introduce new primitives such as

```
matrix ((li,ui),(lj,uj))

update A (i,j) v

A[i,j]
```

These operations leave a lot of room for choosing the internal representation of arrays. In order to achieve constant time access, at the expense of $O(n)$ allocation and update, we will only look at representations that allocate arrays as contiguous chunks of memory. Other researchers have looked at implementations based on trees, where selection and update are both $O(log\ n)$, and where it is possible to have extensible arrays: Ackerman [1] studied implementations based on binary trees, and Thomas [7] studied implementations based on 2-3 trees.

But none of these implementations are adequate in of themselves— they all involve far too much unnecessary copying and unnecessary sequentialization, as we will demonstrate in the next section. Thus, they are always considered along with some major compile-time and/or run-time optimizations to recoup efficiency and parallelism, and these are discussed in subsequent sections.

[3]In Id, the comma is an infix tupling operation, so that the expression "`e1,...,en`" denotes a n-tuple whose components are the values of `e1`, ..., `en` respectively.

3.2.1 Copying and Sequentialization of Update

A direct implementation of the "**update A i v**" operator would be:

- allocate an array with the same index bounds as "**A**",
- copy all elements from "**A**" to the result array, except at location "**i**",
- store value "**v**" in location "**i**" of the result array,
- return the pointer to the result array.

The array selection operation would simply read a memory location at an appropriate offset from the pointer to the array argument.

Example A will suffice to demonstrate that such a direct implementation is grossly inefficient. Here is a solution that allocates an array, and then uses (tail-) recursion to traverse and fill it with the appropriate contents:

```
Def A = {  A = matrix ((1,m),(1,n))
         In
           traverse A 1 1 } ;

Def traverse A i j =
        {  next_A = update A (i,j) (i+j) ;
         In
           If        (j < n) Then traverse next_A i (j+1)
           Else If (i < m) Then traverse next_A (i+1) 1
           Else next_A } ;
```

We use the syntax

```
{ BINDING ... BINDING In EXPRESSION }
```

for blocks, which are like "letrec" blocks in other functional languages, and follow the usual static scoping rules.

We prefer to use the following loop syntax to express the tail-recursions:

```
{  A = matrix ((1,m),(1,n))
 In
   {For i From 1 To m Do
      Next A = {For j From 1 To n Do
                  Next A = update A (i,j) (i+j)
                Finally A}
    Finally A}
```

In the first iteration of the inner loop body, the "A" on the right-hand side refers to its value in the surrounding scope (in this case, the matrix of "nil"s allocated at the top of the block). In each iteration of the loop, the phrase "Next A" binds the value of A for the next iteration. The phrase "Finally A" specifies the ultimate value to be returned at the end of the iteration.

There are two major difficulties in such a program. The first is its profligate use of storage. It is clear that, using a direct implementation of **update**, we would create $(mn+1)$ arrays, of which only one– the final one– is of interest. Each intermediate array carries only incrementally more information than the previous intermediate array.

The second criticism of this program is that it *over-specifies the order of the updates*. In the problem specification, each element can be computed independently of the others. However, because of the nature of the **update** primitive, it is necessary for us to chain all the **update**s involved in producing the final value into a linear sequence.

The necessity to sequentialize the updates also affects program clarity adversely— it is an extra (and unnecessary) bit of detail to be considered by the programmer and reader. Consider a solution for the wavefront problem (Example B):

```
{  A = matrix ((1,m),(1,n))
 In
   {For i From 1 To m Do
      Next A = {For j From 1 To n Do
                  v = If (i == 1) or (j == 1) Then 1
                      Else    A[i-1, j ]
                            + A[i-1,j-1]
                            + A[ i ,j-1] ) ;
                  Next A = update A (i,j) v
                Finally A}
    Finally A}
```

It takes some careful thought to convince oneself that the above program is correct— that the array selections in computing "v" actually read previously computed values and not "nil", the original contents of A. For example, if the recurrence had been specified instead as $A_{i-1,j}+A_{i-1,j+1}+A_{i,j+1}$ (with appropriate boundary conditions), the programmer would have to realize that the j iteration would have to be reversed to count down from n to 1. This is a great departure from the "declarative" nature of the original recurrence specification.

3.2.2 Using Reference Counts to Reduce Storage Requirements

Several researchers have recognized that we can use reference counts to improve the efficiency of the **update** operation. The idea is very simple: assume that associated with each data structure is a number, called its "reference count" (RC), which counts the number of pointers to it that are currently outstanding. The RC of a structure is incremented every time a copy of its pointer is made and decremented every time its pointer is discarded.

If the RC of the argument array is 1 when the **update** operation executes, there can be no other references to the array. The update operation can thus safely be performed *in situ*, by destructively writing the value into the existing array and returning a pointer to the existing array. This is of course *much* cheaper than allocating and filling a new array. This solution has been studied carefully in [1]. Unfortunately, except where the program is written with an artificial sequentialization of array accesses and updates, opportunities for this optimization occur but rarely in a parallel machine.

We must also consider that *every* **update** operation now pays the overhead of checking the RC. Further, the space and time behavior of the program becomes very unpredictable, because whether or not the RC is 1 depends on the particular schedule for processes chosen by the operating system. This can depend, for example, on the current load and configuration of the machine.[4]

In [8], Hudak has proposed a technique called "abstract reference counting", in which a program is analyzed statically to predict the reference counts of arrays at various program points (see also [6]). When the analysis predicts that the reference count of the array argument to an **update** operation will be one, the compiler generates code to perform an update *in situ*.

Hudak's analysis was performed with respect to a *sequential* operational semantics, and relies on the sequential chaining of the collection of **update** operations. In this regard, Hudak reports great success in his experiments. We believe that it will be possible to predict that in our program for Example A, the reference count for each **update** will indeed be one; thus exactly one array will be allocated, and *all* the updates will be done destructively, resulting in a program as efficient (and as sequential) as its FORTRAN counterpart!

Another problem is that the analysis can be sensitive to the order in which the programmer writes his program. Consider a program to compute an array that is identical to a given array `A` except that the `i`'th and `j`'th elements are exchanged:

```
1)    { vj = A[j]
2)     In { vi = A[i]
3)        In { B = update A i vj
4)           In update B j vi }}}
```

Consider a sequential operational semantics that specifies that the bindings of a block are executed before the body of the block. Static analysis may then predict that lines 1 and 2 have been completed before executing line 3, and so the reference count of `A` in line 3 should be 1. Thus the update can be done in place. Similarly, the update in line 4 can also be done in place. But the programmer could easily have written the program with lines 2 and 3 exchanged:

```
1)    { vj = A[j]
```

[4]Maintaining RCs at run time also raises other issues which are beyond the scope of this paper, such as how much additional code/network-traffic there is to maintain RCs; how much contention there is at the RC field of an array amongst all operations on that array; how atomically to increment/decrement the RC field; how to avoid races between increment and decrement operations, *etc.*

```
3)      In {  B = update A i vj
2)          In {  vi = A[i]
4)              In update B j vi }}}
```

The reference count of A in line 3 is no longer 1 because of the outstanding reference in line 2, and so the update in line 3 *cannot* be done in place. The update in line 4 can still be done in place.

Now consider a *parallel* operational semantics for the language. A precise example of such a semantics is given in Section 5, but, for now, imagine that the bindings of a block can be executed in parallel with the body, with sequencing, if any, based only on data dependencies. All four lines of the program are now initiated in parallel. Since there are no data dependencies between lines 2 and 3, their order of execution is unpredictable. Thus, static analysis cannot draw any definite conclusions about the reference count of A in line 3.

3.2.3 Using Subscript Analysis to Increase Parallelism

We have seen that the nature of the **update** primitive requires the programmer to sequentialize the sequence of updates in computing an array. Reference count analysis sometimes determines that these updates may be done in place.

If static analysis could further predict that the subscripts in the sequence of updates were disjoint, then the updates would commute— they could then all be done in parallel. Using such analysis on our program for Example A in Section 3.2.1, the compiler could generate code to perform all the mn updates in parallel.

Subscript analysis has been studied extensively, most notably by Kuck *et al.* at the University of Illinois [11,14] and Kennedy at Rice University [2]. Most of this work was done in the context of vectorizing compilers for FORTRAN. In general, this is an intractable problem, but in the commonly occuring case where the subscripts are of the form $ai + b$ (a and b are constants, i is a loop index), subscript analysis can reveal parallelism. However, there is a significant cost to this analysis, both in terms of compilation speed and in terms of the effort to develop a compiler.

Compared to FORTRAN, subscript analysis is on the one hand easier in functional languages due to referential transparency, but on the other hand more difficult because of dynamic storage allocation.

An example of a program where subscript analysis cannot extract any useful information is a solution to Example C, the Inverse Permutation problem:

```
Def B = {  B = array (1,n)
         In
           For i From 1 To n Do
             Next B = update B A[i] i
           Finally B } ;
```

In order to parallelize the loop, the compiler needs to know about the contents of `A`, such as the fact that it contains a permutation of $1..n$. This is in general too much to ask of compile-time analysis. This situation is not artificial or unusual— it occurs all the time in practical codes, such as in sorting algorithms that avoid copying large elements of arrays by manipulating their indices instead, and in Monte Carlo techniques and Random Walks.

3.3 Discussion

We hope we have convinced the reader of the inadequacy of "fine-grained" functional data structuring mechanisms such as `Cons` and `Update`, especially in a parallel environment. (Some of these problems are solved using the "`make-array`" primitive discussed in the next section.)

Writing programs directly in terms of these primitives does not result in very perspicuous programs— `Cons` requires the programmer continuously to keep in mind the list representation, and `update` requires the programmer to devise a sequential chaining of more abstract operations. In both cases, it is advisable first to program some higher-level abstractions and subsequently to use those abstractions.

Both operators normally involve substantial unnecessary copying of intermediate data structures and substantial unnecessary sequentialization. It was possible to avoid these overheads only when the compiler could be assured that a) reference counts were one, and that b) the subscripts in a chain of updates were disjoint.[5] Automatic detection of these properties does not seem tractable in general.

There is a disquieting analogy with FORTRAN here. Our functional operators force *over-specification* of a problem solution, and static analysis attempts to relax unnecessary constraints. Parallelizing FORTRAN compilers face the same problem, albeit for a different reason (side effects).

4 `Make-Array`: A Bulk Functional Data Structure Operation

Many researchers (notably Keller) have proposed a "bulk" array-definition primitive that treats an array as a "cache" for a function over a rectangular subset of its domain [10,5]. For example, the expression

```
make-array (l,u) f
```

where `(l,u)` is a pair (2-tuple) of integers and `f` is a function, returns an array whose index bounds are `(l,u)`, and whose `i`'th component contains `(f i)`. We will often refer to `f` as the "filling function".

One can think of the array *as a cache* for `f` because for `i` within bounds, `A[i]` returns the same value as `(f i)`, but (hopefully) at significantly lower cost.

[5] Originally, I-structures were functional data structures with these two properties [4].

Higher dimensional arrays may be constructed either by nesting arrays, or by generalizing the primitive. Thus,

```
make-matrix ((li,ui),(lj,uj)) f
```

produces a matrix where the `(i,j)`'th component contains `f (i,j)`. The first argument is a pair of pairs of integers, and specifies the index bounds of the matrix.

4.1 Example A

We can now readily see the solution for Example A:

```
Def f (i,j) = i + j;

Def A = make-matrix ((1,m),(1,n)) f
```

which is concise and elegant and does not pose any serious problem for an efficient, parallel implementation.

4.2 Strictness of make-array

Before moving on to the remaining examples, it is worth noting that **make-array** need not be strict, *i.e.*, the array may be "returned" before any of the component values have been filled in.

An eager implementation (such as a dataflow implementation) may behave as follows: the bounds expression is first evaluated, and storage of the appropriate size allocated. Then, n independent processes are initiated, computing `(f 1)`, ..., `(f n)` respectively—each process, on completion, writes into the appropriate location in the array. Meanwhile, a pointer to the array is returned immediately as the result of the **make-array** expression. Some synchronization mechanism is necessary at each array location, so that a consumer that tried to read some `A[i]` while it is still empty is made to wait until the corresponding `(f i)` has completed. (Later, we shall see how I-structures provide this synchronization.)

A lazy implementation of **make-array** may behave as follows: the bounds expression is first evaluated, and storage of the appropriate size allocated. Each location `A[i]` is then loaded with the suspension for `(f i)`, and the pointer to the array is returned. A subsequent attempt to read `A[i]` will trigger the evaluation of the suspension, and the value will overwrite the suspension.

This kind of non-strictness permits a "pipelined" parallelism in that the consumer of an array can begin work on parts of the array while the producer of the array is still working on other parts. Of course, even the **Cons** and **Update** operators of Section 3 could benefit from this type of non-strictness.

4.3 Example B (Wavefront)

A straightforward solution that comes to mind immediately for the wavefront example is:

```
Def f (i,j) = If (i == 1) or (j == 1) Then 1
              Else   f (i-1,  j )
                   + f (i-1, j-1)
                   + f ( i , j-1) ;

Def A = make-matrix ((1,m),(1,n)) f ;
```

But this is extremely inefficient because "`f (i,j)`" is evaluated repeatedly for each `(i,j)`, not only to compute the `(i,j)`'th component, but also during the computation of every component to its right and below. (This is the typical exponential behavior of a recursively defined Fibonacci function.)

The trick is to recognize that the array is a "cache" or "memo" for the function, and to use the array itself to access already-computed values. This can be done with a recursive definition for `A`:

```
Def f X (i,j) = If (i == 1) or (j == 1) Then 1
                Else   X[i-1,  j ]
                     + X[i-1, j-1]
                     + X[ i , j-1] ;

Def g = f A ;
Def A = make-matrix ((1,m),(1,n)) g;
```

Here, the function `f` is a curried function of two arguments, a matrix and a pair of integers. By applying it to `A`, `g` becomes a function on a pair of integers, which is a suitable argument for `make-matrix`. The function `g`, in defining `A`, carries a reference to `A` itself, so that the computation of a component of `A` has access to other components of `A`.

In order for this to achieve the desired caching behavior, the language implementation must handle this correctly, *i.e.,* the `A` used in `g` must be the *same* `A` produced by `make-matrix` and *not* a new copy of the definition of `A`.

Assuming the language implementation handles this correctly, the main inefficiency that remains is that the `If-Then-Else` is executed at every location.

Note that in a general recursive definition, it will be impossible to predict statically in what order the components must be filled to satisfy the dependencies, and so a compiler cannot "pre-schedule" the computation of the components of an array. Thus, any implementation necessarily must use some of the dynamic synchronization techniques mentioned in Section 4.2.

4.4 Example C (Inverse Permutation)

Unfortunately, `make-array` does not do so well on Example C (Inverse Permutation):

```
Def find i j = If (B[j] == i) Then j
               Else find i (j+1) ;

Def f i = find i 1 ;

Def A = make-array (1,n) f ;
```

The problem is that each (f i) that is responsible for filling the i'th location of A needs to search B for the location that contains i, and this search must be linear. Thus, the cost of the program is $O(n^2)$.

It is possible to come up with a slightly different array primitive that addresses this problem. Consider

```
make-array-1 (l,u) f
```

where each (f i) returns (j,v), so that A[j] = v, *i.e.*, the "filling function" f is now responsible for computing not only a component value, but also its index.[6] Example C may now be written:

```
Def f i = B[i],i ;

Def A = make-array-1 (1,n) f ;
```

Of course, if B does not contain a permutation of $1..n$, a run-time error must be detected—either two (f i)'s will attempt to write the same location, or some (f i) will attempt to write out of bounds.

Note that this new primitive, **make-array-1**, no longer has the simple and elegant characterization of **make-array** as being a "cache" for the filling function— the relation between the array and the filling function is no longer straightforward. In supplying a filling function for **make-array-1**, the programmer must now take care that the indices computed by f form a permutation of $1..n$.

4.5 Example D (Shared Computation)

A straightforward solution to the shared computation problem may be written as follows:

```
Def fh i = f (h i) ;
Def gh i = g (h i) ;

Def A = make-array (1,n) fh ;
Def B = make-array (1,n) gh ;
```

[6] We first heard this solution independently from David Turner and Simon Peyton-Jones, in a slightly different form— instead of having a filling function f, they proposed an association-list of index-and-value pairs. This solution is also mentioned by Wadler in [17].

for A and for B.

One possible way out is first to cache the values of (h i) in an array C:

```
Def C = make-array (1,n) h ;

Def fh i = f C[i] ;
Def gh i = g C[i] ;

Def A = make-array (1,n) fh ;
Def B = make-array (1,n) gh ;
```

The drawback is the overhead of allocating, writing, reading and deallocating the intermediate array C.

To regain the sharing, one could imagine the following scenario performed by an automatic program transformer. The two make-arrays are expanded into, say, two loops. Recognizing that the loops have the same index bounds, they are fused into a single loop. Within the resulting loop, there will be two occurrences of (h i); this common sub-expression can then be eliminated.

We believe that this scenario is overly optimistic. It is very easy to modify the example very slightly and come up with something for which an automatic program transformer would have no chance at all— for example, by changing or displacing the index bounds of one array, or by having a sharing relationship that is not one-to-one, *etc.*

4.6 Discussion

Any functional data-structuring primitive must specify two things: the storage to be allocated and the contents of that storage. For example, Cons e1 e2 specifies a two-cell storage allocation, together with the contents of the two cells.

For a large data structure such as an array, it is obviously not feasible to enumerate expressions specifying the contents of all the components. Thus, functional primitives for large data structures must specify a regular way to *generate* the contents. Thus, make-array takes the "filling" parameter f, and sets up n independent processes, with the i'th process responsible for computing and filling the i'th location.

We saw two problems with this process structure. First, this early binding of the i'th process to the i'th location could not handle things like the inverse permutation. Second, there was no convenient way to express shared computation between the filling processes.

The variant make-array-1 solved the first problem, by leaving it up to each of the i processes to decide which index j it was responsible for; but it still did not address the issue of shared computation, which could only be performed with the overhead of constructing intermediate arrays or lists. In recent correspondence with us, Phil Wadler has conjectured that, using the version of make-array-1 that uses association lists of index-and-value pairs together with his "list-less transformer" [16], these problems may indeed be solved without any overhead of intermediate lists. We have yet to investigate the viability of this approach.

All the examples we have seen are quite small and simple; even so, we saw that the first, straightforward solution that came to mind was in many cases quite unacceptable, and that the programmer would have to think twice to achieve any efficiency at all. The complications that were introduced to regain efficiency had nothing to do with improving the algorithms— they were introduced to get around the language limitations.

We are thus pessimistic about relying on a fixed set of functional data structuring primitives. We have encountered situations where the problems illustrated above do not occur in isolation— recursive definitions are combined with shared computations across indices and across arrays. In these situations, writing efficient programs using functional array primitives has proven to be very difficult, and is almost invariably at the expense of program clarity. Perhaps, with so many researchers currently looking at this problem, new functional data-structuring primitives will emerge that will allow us to revise our opinion.

5 I-Structures

In the preceding discussion of functional data structures, we saw that the source of inefficiency is the fact that the various primitives impose too rigid a structure on the processes responsible for filling in the components of the data structure. Imperative languages do not suffer from this drawback, because the allocation of a data structure (variable declaration) is decoupled from the filling-in of that data structure (assignment). But imperative languages, with unrestricted assignments, complicate parallelism because of timing and determinacy issues. I-structures are an attempt to regain that flexibility without losing determinacy.

In the Section 5.1 we present the operations that define I-structures with an informal and intuitive explanation.

In Sections 5.2 and 5.3, respectively, we present a precise operational semantics for the functional subset of Id, and for Id with I-structures. This operational semantics is defined by a confluent set of rewrite rules and a reduction strategy which may be loosely described as "Parallel Call-by-Value" or *the dataflow rule of computation*. It provides a basis against which to evaluate the efficiency and parallelism of I-structures.

The rewrite rules and the associated reduction strategy of Id are unusual compared to what experts in functional language may be familiar with. To begin with, they describe exactly what computations are shared, an issue that can (and usually is) left unspecified for other functional languages. This allows us then, even for the functional subset of Id, to describe the dataflow rule of computation, which is a parallel normalizing rule. Further, this specification of sharing (or exactly how many times an expression is evaluated) is necessary to preserve confluency in a language with I-structures.

Because of these subtleties, we invite the reader to study Sections 5.2 and 5.3 carefully, even though they may be skipped at a first reading.

5.1 I-structure operations

One can think of an I-structure as a special kind of array, each of whose components may be written no more than once. To augment a language with I-structures, we introduce three new constructs.

An I-structure is allocated by the expression

```
array (l,u)
```

which allocates and returns an "empty" array whose index bounds are `(l,u)`. I-structures are first-class values, and they can contain other I-structures, functions, *etc.* We can simulate multi-dimensional arrays by nesting I-structures, but for efficiency reasons Id also provides primitives for directly constructing multi-dimensional I-structures:

```
matrix ((li,ui),(lj,uj))
```

and so on.

A given component of an I-structure `A` may be assigned (written) no more than once, using a "constraint statement":

```
A[i] = v
```

Operationally, one thinks of this as assigning, or storing the value `v` into the `i`'th location of array `A`. It is a run-time error to write more than once into any I-structure location— the entire program is considered to be in error.

Syntactically, the assignment statement appears intermixed with the bindings in a block.

A component of an I-structure `A` may be selected (read) using the expression

```
A[i]
```

This expression returns a value only after the location becomes non-empty, *i.e.,* after some other part of the program has assigned the value.

There is no test for "emptiness" of an array location. These restrictions— write-once, deferred reads, and no test for emptiness— ensure that the language remains *determinate*; there are no read-write races. Thus the programmer need not be concerned with the timing of a read relative to a write. All reads of a location return a single, consistent value, albeit after an arbitrary delay.

Semantically, one can think of each location in an I-structure as containing a *logical term*. Initially, the term is just a logic variable— it is completely unconstrained. Assignment to that location can be viewed as a refinement of, or constraint on the term at that location. This is what motivates our calling it a "constraint statement". The single-assignment rule is sufficient to preclude inconsistent instantiations of the initial logic variable. Of course, the single-assignment rule is not a *necessary* condition to avoid inconsistent instantiations. We could take the view that assignment is really ***unification***, and then multiple writes would be safe so long as the values unify. Id does not currently take this view, for efficiency reasons.

Some machine-level intuition: Conceptually, I-structures reside in an "I-structure memory" unit. When an allocation request arrives, an array is allocated in free space, and a pointer to this array is returned. Every location in the array has an extra bit that designates it as being "empty".

Every request to read a location of an I-structure (*i.e.*, a selection) is accompanied by a "tag", which can be viewed as the name of the continuation that expects the result. The I-structure controller checks the "empty" bit at that location. If it is not empty, the value is read and sent to the continuation. If the location is still empty, the controller simply queues the tag at that location.

When a request to store a value in a location of an I-structure arrives (*i.e.*, an assignment), The I-structure controller checks the "empty" bit at that location. If it is empty, the value is stored there, the bit is toggled to "non-empty", and if any tags are queued at that location, the value is also sent to all those continuations. If the location is not empty, the controller generates a run-time error.

5.2 Id and its Operational Semantics: the Functional Subset

In this section we give the reader a feel for the parallelism in dataflow execution of Id. Previously, this has only been described in terms of dataflow graphs, the "machine language" of the dataflow machine. Here we describe it in terms of a confluent set of rewrite rules that capture precisely the behaviour of dataflow graphs. We consider this description more abstract and probably easier to understand for those already familiar with rewrite rules. Our description here is necessarily brief— the reader is referred to [13,12,15] for more comprehensive treatments.

The reader is invited to read this section carefully, as the rewrite rules incorporate many subtle differences from other execution models for functional languages that we are aware of.

5.2.1 Syntax

Here is an abbreviated syntax for Id:

```
PROGRAM ::= FUN-DEF ";" ... ";" FUN-DEF ";" EXPRESSION

FUN-DEF ::= "Def" IDENTIFIER ARG ... ARG "=" EXPRESSION

ARG     ::= IDENTIFIER

EXPRESSION ::=
            CONSTANT | IDENTIFIER
```

```
            "If" EXPRESSION "Then" EXPRESSION "Else" EXPRESSION
            EXPRESSION EXPRESSION
            BLOCK

BLOCK   ::= "{" BINDING ";" ... ";" BINDING "In" EXPRESSION "}"

BINDING ::= IDENTIFIER "=" EXPRESSION
```

i.e., a program is a list of function definitions, and a main expression. The arguments in function definitions are curried, and the definitions may be recursive and mutually recursive. Blocks are analogous to the "letrec" construct in other functional languages; the bindings may be recursive and mutually recursive, they follow the usual static scoping rules, and the order of bindings in a block is not significant.

For simplicity of exposition here, the bindings in blocks do not include syntactic function definitions (there are no arguments), so we do not have to worry about dynamically handling free variables. This is not a serious restriction— we use the technique of lambda-lifting [9] to "compile out" internal function definitions, leaving programs in the above standard form.

Since all function definitions occur only at the top (program) level, one can think of the set of function definitions as a set of combinator definitions, or rewrite rules.

We define the "arity" of a function identifier as the number of arguments in its definition. This is a *syntactic*, not semantic attribute. Thus the following two definitions

```
Def f a   = plus (sqr a)   ;
Def g a b = plus (sqr a) b ;
```

define semantically equivalent functions of two arguments, but have arities 1 and 2 respectively, by virtue of their different syntactic definitions.

Though arity is a syntactic attribute, it determines which expressions are redexes (rewritable), which in turn affects the definition of "normal forms", or answers. Thus `(f 5)` is a redex, and so is not in normal form, whereas `(g 5)` is not a redex, and *is* in normal form. This "pattern matching" based on the syntactic definition is fairly standard in rewrite-rule systems.

We have the usual conventions that

- Applications associate to the left
- Infix expressions "`x op y`" are syntactic sugar for "`opfn x y`", for some known set of infix operators such as `+`.
- Parentheses are used for grouping

5.2.2 Values

The purpose of a program is to produce a *value* for the main expression. A value is an expression in "normal form". We will use the generic symbol "V" to denote a value.

Values include all constants, such as numbers, and the booleans **true** and **false**. Values also include partial applications of the form

```
f x1 ... xj
```

where the xi's are identifiers or values (not general expressions), and the arity of f is greater than j. Thus, for example, if g is function of arity 2, then g and (g x) are values, but (g x y) is not. Similarly, if there is a binding

```
a = EXPRESSION
```

then the expression "a" is not a value— its arity (0) is satisfied, and so it is not a partial application.

These technical definitions of values may seem an over-specification for a functional language; but they are necessary in a language with I-structures, where we must specify, for each expression, things like its sharing behavior, how many times it is evaluated, *etc.*

These values have a one-to-one correspondence with "tokens" in the dataflow machine.

5.2.3 Machine States and Termination

A machine state is described as follows:

```
E ; B ; ... ; B
```

where E is an expression and B's are bindings. E is called the "main" expression. There may be zero or more bindings, and their order is not significant.

The initial state of the machine is just

```
E
```

i.e., an expression and no bindings, where E is the main expression of the program.

The machine is in its final state when no rewrite rules can be applied *anywhere* in the machine, *i.e.*, it is of the form

```
V ; IDENTIFIER = V ; ... ; IDENTIFIER = V
```

(all the B's are quiescent). Note that the main expression may reduce to a value *before* all the bindings become quiescent.[7]

[7]On our dataflow machine this phenomenon is reflected in the sometimes unnerving behavior that an answer may by printed long before termination is reported!

5.2.4 Rewrite Rules

Each rewrite rule identifies a sub-expression somewhere in the machine state and shows what that sub-expression reduces to. The sub-expression may occur in **E**, or in the right-hand sides of any of the **B**'s; thus one should imagine all the components of the machine state executing in parallel.

We use the notation

```
    (e)
    E ; B ; ... ; B
```

to focus our attention on a sub-expression "**e**" that occurs somewhere in **E** or somewhere in one of the right-hand sides of the **B**'s, *but not within the* **Then** *or* **Else** *arms of a conditional expression.* Such an "**e**" is called a *redex*.

The dataflow rule requires *all* such redexes to be reduced ultimately. Of course, in a real implementation only a subset of redexes can be rewritten at any step, and so some form of fair scheduling is required in order to produce answers.

1) Identifiers may be substituted by *values* (not expressions!) that they are bound to:

```
    (x)
    E ; B ; ... ; x = V ; ... ; B
==>
    (V)
    E ; B ; ... ; x = V ; ... ; B
```

The fact that substitution must wait for the expressions to reduce to values motivates our characterizing it as a "call-by-value" semantics.

2) Conditionals:

```
    (If true Then E1 Else E2)
    E ; B ; ... ; B
==>
    (E1)
    E ; B ; ... ; B
```

and similarly for the **false** case.

3) Partial Applications (arity not satisfied):

Suppose the program has a definition

```
    Def f x1 ... xn = Ebody ;
```

Then

```
    (f Earg1 ... Eargj)
    E ; B ; ... ; B
==>
    (f x1' ... xj')
    E ; B ; ... ; B ; x1' = Earg1 ; ... ; xj' = Eargj
```

where j < n, and **x1'** through **xj'** are new identifiers.

As a result of this transformation, the partial application now conforms to the definition of a value, and so may be passed around and shared (for example, applied to different remaining arguments). However, by Rule **1**, the shared arguments **x1** through **xj** are evaluated exactly once.

4) Full Applications (arity satisfied):

Suppose the program has a definition

```
    Def f x1 ... xn = Ebody ;
```

Then

```
    (f Earg1 ... Eargn)
    E ; B ; ... ; B
==>
    (Ebody')
    E ; B ; ... ; B ; x1' = Earg1 ; ... ; xn' = Eargn
```

where **x1'** through **xn'** are new identifiers and **Ebody'** is an appropriately alpha-renamed version of **Ebody**. Note the parallel behavior: the body of the function is evaluated in parallel with the arguments, but by Rule **1**, the arguments are not substituted in the body until they have reduced to values. This is what motivates the name "Parallel Call-By-Value", and demonstrates how non-strict functions can return values before receiving argument values.

5) Blocks:

```
    ({ B1 ; ... ; Bn IN E0 })
    E ; B ; ... ; B
==>
    (E0')
    E ; B ; ... ; B ; B1' ; ... ; Bn'
```

where **E0'** and **B1'** through **Bn'** use new identifiers for the identifiers bound in the block.

5.2.5 An Example

Consider the following program:

```
Def twice f x = f (f x) ;

twice (plus 5) (2 * 3)
```

Here is a possible rewrite sequence (in each step we have underlined redex(es) that we have chosen to rewrite):

```
    twice (plus 5) (2 * 3)
    ^^^^^^^^^^^^^^^^^^^^^^
==>
    f1 (f1 x1) ; f1 = plus 5 ; x1 = 2 * 3
    ^^  ^^                          ^^^^^
==>
    plus 5 (plus 5 x1) ; f1 = plus 5 ; x1 = 6
                   ^^
==>
    plus 5 (plus 5 6) ; f1 = plus 5 ; x1 = 6
            ^^^^^^^^
==>
    plus 5 11 ; f1 = plus 5 ; x1 = 6
    ^^^^^^^^^
==>
    16 ; f1 = plus 5 ; x1 = 6
```

5.2.6 Discussion

It is important to note that Id, with these operational semantics, supports non-strict functions, *but it is eager, not lazy.* Given an application (`f x`), the body of `f` is invoked in parallel with the evaluation of `x`. If `f` does not need `x`, it may return a result immediately. However, `x` is still computed; it is not left up to `f` to "demand" `x`. Unfortunately, in the literature, non-strictness is often equated with laziness.

The effect of eager evaluation in producing answers becomes an issue only when an argument to a function is not used, because resources are used in evaluating it. An extreme case of this occurs when such an argument computation diverges. Consider the following program:

```
Def diverge x = diverge x ;

Def f y = 5 ;

f (diverge 1)
```

A correct implementation of Id should print the value `5` even though the program does not terminate. This issue does not arise in any of the other examples discussed in this paper.

Two aspects of the rewrite rules deserve close scrutiny: A) the rewrite rules have been carefully constructed to ensure that arguments are evaluated exactly once, and B) there is a qualification that the redex not be within the arms of an `If-Then-Else`. In the functional subset of Id, these affect only the efficiency of the reduction— they avoid wasting machine resources— and do not affect the outcome of the program. When Id is augmented with I-structures, these restrictions become semantically necessary.

5.3 Operational Semantics of Id with I-Structures

5.3.1 I-structure Values

We first introduce a notation for I-structure values.

```
    <X1, ..., Xu>
```

denotes an I-structure, where the `Xi`'s are identifiers. One can think of an I-structure value as a pointer to an array in I-structure memory, with `X1...Xu` the names of the locations in the array.

5.3.2 Rewrite Rules for I-structure Operations

There are two new rewrite rules for I-structures:

```
    array (v1,v2)
=>
    <Xv1, ..., Xv2>
```

where `v1` and `v2` are integer values, and `Xv1, ..., Xv2` are new identifiers. Note that this rewrite rule has a characteristic never found in functional languages, but common in logic programming languages— new variables are introduced on the right-hand side.

The rule

```
    <X1, ..., Xu>[V]
=>
    XV
```

specifies the selection of one of the locations of the I-structure value. For a minor technical reason having to do with termination, the identifiers inside an I-structure value are never substituted by their values.

5.3.3 An Example

Consider the following program:

```
Def pair x y = {  a = array (0,1) ;
                  a[0] = x ;
                  a[1] = y
                In
                  a } ;

pair 5 (pair 6 7)
```

Here is a possible rewrite sequence (some of the machine state descriptions take two lines, and we have omitted introducing identifiers for function arguments that are already values):

```
    pair 5 (pair 6 7)
    ^^^^^^^^^^^^^^^^^
==>
    {a=array(0,1); a[0]=5 ; a[1]=y1 IN a} ; y1 = pair 6 7
                                                 ^^^^^^^^
==>
    {a=array(0,1); a[0]=5 ; a[1]=y1 IN a} ;
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
        y1={a=array(0,1); a[0]=6 ; a[1]=7 IN a}
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
==>
    a1 ; a1=array(0,1) ; a1[0]=5 ; a1[1]=y1 ;
            ^^^^^^^^^^
        y1=a2; a2=array(0,1); a2[0]=6 ; a2[1]=7
                  ^^^^^^^^^^
==>
    a1 ; a1=<X0,X1> ; a1[0]=5 ; a1[1]=y1 ;
    ^^                ^^        ^^
        y1=a2; a2=<Y0,Y1> ; a2[0]=6 ; a2[1]=7
           ^^               ^^        ^^
==>
    <X0,X1> ; a1=<X0,X1> ; <X0, X1>[0]=5 ; <X0,X1>[1]=y1 ;
                           ^^^^^^^^^^^     ^^^^^^^^^^ ^^
        y1=<Y0,Y1>; a2=<Y0,Y1> ; <Y0,Y1>[0]=6 ; <Y0,Y1>[1]=7
                                 ^^^^^^^^^^     ^^^^^^^^^^
==>
    <X0,X1> ; a1=<X0,X1> ; X0=5 ; X1=<Y0,Y1> ;
        y1=<Y0,Y1>; a2=<Y0,Y1> ; Y0=6 ; Y1=7
```

Note that the resulting I-structure value `<X0,X1>` appears before termination. On the dataflow machine, an I-structure descriptor (pointer) is printed before termination. If the user subsequently queries the zero'th element (say) of the I-structure, the component value will be fetched (corresponding to dereferencing `X0` to the value 5).

The astute reader may ask: since computation may go on after a value is printed, what is to be made of a run-time error that then occurs? There is no problem of determinacy here— a program with a run-time error is considered to have the value "$\top$", the "inconsistent" element at the top of the semantic domain. A value printed out before termination is to be considered as an approximation to the final value; a run-time error only refines this approximation to $\top$.

5.4 The Programming Examples

The assignment statement is the simplest form of a constraint statement, and this can be generalized into compound constraint statements.

By embedding assignments in loops, we can now also have loops that behave purely as constraint statements, by omitting the "**Finally e**" clause.

A function "**f**" may not return any value of interest— it may contain only constraint statements. A call to such a function is itself a constraint statement, using the syntax

```
Call f x
```

Constraint statements— assignments, constraint loops, constraint calls— appear intermixed with the bindings in a block or loop-body.

Let us now see how our programming examples would be expressed using Id with I-structures.

5.4.1 Example A

The first example is straightforward:

```
{ A = matrix ((1,m),(1,n)) ;
  {For i From 1 To m Do
     {For j From 1 To n Do
        A[i,j] = i + j }}
In
  A }
```

Recall that the loop is a *parallel* construct, so in the above program, the loop bodies can be executed in any order— sequentially forwards, as in FORTRAN, or all in parallel, or even sequentially backwards!

The matrix **A** may be returned as the value of the block as soon as it is allocated. Meanwhile, $m \times n$ loop bodies execute in parallel, each filling in one location in **A**. Any consumer that tries to read **A[i,j]** will get its value as soon as the corresponding loop body completes.

5.4.2 Example B (Wavefront)

```
{  A = matrix ((1,m),(1,n)) ;
   {For i From 1 To m Do
      A[i,1] = 1 } ;
   {For j From 2 To n Do
      A[1,j] = 1 } ;
   {For i From 2 To m Do
      {For j From 2 To n Do
         A[i,j] = A[i-1,j] + A[i-1,j-1] + A[i,j-1] }}
In
   A }
```

The matrix **A** may be returned as the value of the expression as soon as it is allocated. Meanwhile, *all* the loop bodies are initiated in parallel, but some will be delayed until the loop bodies to their left and top complete. Thus a "wavefront" of processes fills the matrix.

Note that we do not pay the overhead of executing an **If-Then-Else** expression at each index, as in the functional solution.

It is worth emphasizing again that loops are *parallel* constructs. In the above example, it makes no difference if we reverse the index sequences:

```
{For i From m Downto 2 Do
   {For i From n Downto 2 Do ...}}
```

The data dependencies being the same, the order of execution would be the same. This is certainly not the case in imperative languages such as FORTRAN.

5.4.3 Example C (Inverse Permutation)

```
{  A = array (1,n) ;
   {For i From 1 To n Do
      A[B[i]] = i }
In
   A }
```

The array **A** may be returned as the value of the expression as soon as it is allocated. Meanwhile, all the loop bodies execute in parallel, each filling in one location. If **B** does not contain a permutation of 1..n, then a run-time error will arise, either because two processes tried to assign to the same location or because some process tried to write out of bounds.

5.4.4 Example D (Shared Computation)

```
{  A = array (1,n) ;
   B = array (1,n) ;
   {For i From 1 To n Do
      z = h i ;
      A[i] = f z;
      B[i] = g z }
In
   A,B }
```

The arrays A and B may be returned as the value of the expression as soon as they are allocated. Meanwhile, all the loop bodies execute in parallel, each filling in two locations, one in A and the other in B. In each loop body, the computation of (h i) is performed only once.

6 Using I-structures to Implement Array Abstractions

From the point of view of programming methodology, it is usually desirable for the programmer first to implement higher-level array abstractions and subsequently to use those abstractions.

6.1 Functional Array Abstractions

As a first example, we can implement the functional **make-array** primitive:

```
Def make-array (l,u) f = {  A = array (l,u) ;
                            {For i From l To u Do
                               A[i] = f i }
                         In
                            A } ;
```

Note that there is all the parallelism we need in this implementation. The array A can be returned as soon as it is allocated. Meanwhile, all the loop bodies execute in parallel, each filling in one component. Any consumer that attempts to read a component will get the value as soon as it is filled. It is likely that even if we stick to a purely functional language and supply **make-array** as a primitive, some such I-structure-like implementation mechanism will be necessary to achieve this parallelism.

Similarly, here is an efficient, parallel implementation for **make-matrix**:

```
Def make-matrix ((li,ui),(lj,uj)) f =
                        {  A = matrix ((li,ui),(lj,uj)) ;
```

```
{For i From li To ui Do
   {For j From lj To uj Do
      A[i,j] = f (i,j) }}
In
  A } ;
```

A functional vector sum:

```
Def vector_sum A B = {  l,u = bounds A ;
                        C = array (l,u) ;
                        {For i From l To u Do
                              C[i] = A[i] + B[i] }
                      In
                        C } ;
```

Again, the solution has all the parallelism we need. The array `C` is returned as soon as it is allocated. Meanwhile, independent processes execute in parallel, each computing one sum and storing it in one location in `C`.

The functional **make-array-1** primitive:

```
Def make-array-1 (l,u) f =
                         {  A = array (l,u) ;
                            {For i From l To u Do
                               j,v = f i ;
                               A[j] = v }
                          In
                            A } ;
```

A primitive to make two arrays in parallel:

```
Def make-two-arrays (l,u) f =
                         {  A = array (l,u) ;
                            B = array (l,u) ;
                            {For i From l To u Do
                               va,vb = f i ;
                               A[i] = va ;
                               B[i] = vb }
                          In
                            A,B } ;
```

We leave it as an exercise for the reader to use **make-two-arrays** to produce an elegant solution to the shared computation problem (Example D).

It is clear that, with I-structures in the language, it is straightforward for the programmer to implement any desired functional array abstractions— the solutions are perspicuous, efficient, and there is no loss of parallelism.

6.2 Non-Functional Array Abstractions

It has been our experience that functional abstractions are not the only ones that lead to compact, elegant programs. Consider the following (non-functional) "array-filling" abstraction:

```
Def fill A ((li,ui),(lj,uj)) f =
                          {For i From li To ui Do
                             {For j From lj To uj Do
                                  A[i,j] = f (i,j)}} ;
```

which fills a rectangular region of the given matrix A. Our wavefront program can then be written as follows:

```
{  A = matrix ((1,m),(1,n)) ;
   border   (i,j) = 1 ;
   interior (i,j) = A[i-1,j] + A[i-1,j-1] + A[i,j-1] ;
   Call fill A ((1,m),(1,1)) border ;
   Call fill A ((1,1),(2,n)) border ;
   Call fill A ((2,m),(2,n)) interior
In
   A }
```

Of course, for more efficiency, we could define special abstractions for filling in horizontal or vertical regions:

```
Def fill_col  A ((li,ui),j) f = {For i From li To ui Do
                                      A[i,j] = f (i,j)} ;

Def fill_row  A (i,(lj,uj)) f = {For j From lj To uj Do
                                      A[i,j] = f (i,j)} ;
```

and use them to fill the borders of our matrix.

7 Limitations of I-structures

While we believe that I-structures solve some of the problems that arise with functional data structures, we have frequently encountered another class of problems for which they still do not lead to efficient solutions.

Consider the following problem: we are given a very large collection of generators (say a million of them), each producing a number. We wish to compute a frequency distribution (histogram) of these values in, say, 10 intervals. An efficient parallel solution should allocate an array of 10 "accumulators" initialized to 0, and execute as many generators as it can in parallel. As each generator completes, its result should be classified into an interval j, and the j'th accumulator should be incremented. It does not matter in what order the accumulations are performed, so there are no serious determinacy issues, except the following synchronization requirement: there is a single instant when the resulting histogram is ready (*i.e.*, available to consumers)— it is ready when all the generators have completed. To avoid indeterminacy, no consumer should be allowed to read any location of the histogram until this instant.

A second example: In a system that performs symbolic algebra computations, consider the part that multiplies polynomials. A possible representation for the polynomial

$$a_0 + a_1x + a_2x^2 + a_3x^3 \ldots + a_nx^n$$

would be an array of size $n+1$ containing the coefficients a_0, ..., a_n. To multiply two polynomials A and B of degree n together, we need first to allocate an array of size $2n$, with each location containing an "accumulator" initialized to 0; then, for each j, initiate $(j+1)$ processes to compute $a_0 \times b_j$, $a_1 \times b_{j-1}$, ..., $a_j \times b_0$; as each of these processes completes, its result should be added into the j'th accumulator. The order of the accumulation at any index does not matter.

The synchronization requirement here is more complex. A consumer for a location in the result array may read it as soon as the $j+1$ processes attached to it have completed; this may occur before other locations are ready. Contrast this with the histogram example where the entire array became available to consumers at a single instant.

These problems cannot be solved efficiently either with any of the functional data structures that we have seen so far, or with I-structures. There are two fundamental problems to be addressed:

1. How to model the accumulators. With I-structures and functional data structures, once a location in an array has a value, it cannot be updated at all, even though the update occurs in a safe, structured manner.

2. How to express the termination of the accumulation. In the histogram example, the termination was a global condition. In the polynomial example, termination is tied to each location.

We are currently studying some proposed solutions to this problem.[8]

8 Conclusion

In this paper, we have studied the issue of data structures for parallel computing. We saw that with functional data structures, it can be difficult simultaneously to achieve efficiency, parallelism, and program clarity. We showed that I-structures go a long way towards solving this problem.

[8]In [17], Wadler has proposed yet another functional array operation to handle such "accumulation" problems. This construct combines an association-list of index-and-value pairs, together with a reduction operator to specify the array. We do not yet know what are the implementation issues for this construct.

Further, it is clear that whether or not I-structures are available in the programming language, something like I-structures are needed at the machine level to *implement* functional primitives such as **make-array**, in order that they have maximum parallelism— the read/write synchronization mechanism seems fundamental.

The introduction of any non-functional feature (such as I-structures) into a functional language is not without cost— the language loses referential transparency, with all the attendant implications on the ability to reason about programs, do program transformations for optimization, *etc.* In the case of I-structures, the loss of referential transparency is evident— this program

```
{ a = array (1,10)
In
  a, a }
```

is not semantically equivalent to this program

```
array (1,10), array (1,10)
```

Even so, it is still much easier to reason about programs with I-structures than it is to reason about programs in unconstrained imperative languages, because of the absence of timing issues.

A functional language with I-structures can be made referentially transparent by adopting a "relational" syntax (like logic programming languages) rather than a functional one. Referential transparency is lost in Id because the "**array**" construct allocates an array without naming it. To fix this, we first replace it with a new construct called "**array_bounds**". Array allocation is then achieved by the constraint statement:

```
array_bounds (x) = (l,u)
```

which instantiates "**x**" to be an array with bounds "**(l,u)**". The array is thus not allocated anonymously.

But this is not enough; functional abstraction still allows us to produce anonymous arrays:

```
Def alloc (l,u) = {  array_bounds (x) = (l,u)
                     In x } ;
```

To prevent this, abstraction (indeed all constructs) must be converted to a relational form. For example, a procedure cannot return a value explicitly; rather, it must take an additional argument which it instantiates to the returned value. The **alloc** procedure would be thus be written as follows:

```
Def rel_alloc (l,u) x = { array_bounds (a) = l,u ;
                          x = a }
```

For example, the invocation "`rel_alloc (1,10) a`" will instantiate "`a`" to an array of size 10. Further, to specify that "`a`" is a "place-holder" argument rather than a value passed to `rel_alloc`, we must annotate it appropriately, say with "^". The invocation must therefore be written as "`rel_alloc (1,10) ^a`".

By adopting this annotated relational syntax, we believe that we *could* achieve referential transparency, at the cost of complicating the syntax considerably. We are unconvinced that this is a useful thing to do.

Because I-structure operations compromise referential transparency, as a matter of programming style we strongly encourage the programmer to use only functional abstractions wherever possible. A good Id programmer will separate the program into two parts— a part that defines convenient functional data-structure abstractions in terms of I-structures, and the rest of the program that uses only those abstractions and does not explicitly use I-structures. The latter part of the program is then purely functional, and amenable to all the tools available to manipulate functional languages.

However, to permit efficient parallel implementation of those functional data structure abstractions, I-structures are indispensible, and must be available in the language. As we have argued above, it does not seem possible to define new abstractions entirely in terms of functional primitives without loss of efficiency and parallelism; but as shown in Section 6, I-structures make this absolutely straightforward.

Acknowledgements: Vinod Kathail, who wrote the first Id compiler, brought to light many of the subtle issues on copying and parallelism, and was instrumental in clarifying our understanding of rewrite rules. The current Id compiler, written by Ken Traub, has been invaluable in testing our ideas. K. Ekanadham of IBM first explored the use of I-structures to program high-level abstractions.

References

[1] William B. Ackerman. *A Structure Memory for Data Flow Computers.* Master's thesis, Technical Report TR-186, MIT Lab. for Computer Science, Cambridge, MA 02139, 1978.

[2] J.R. Allen and K. Kennedy. *PFC: A Program to convert FORTRAN to Parallel Form.* Technical Report MASC-TR82-6, Rice University, Houston, TX, March 1982.

[3] Arvind and David Ethan Culler. Dataflow architectures. In *Annual Reviews in Computer Science*, pages 225–253, Annual Reviews Inc., Palo Alto, CA, 1986.

[4] Arvind and Robert E. Thomas. *I-structures: An Efficient Data Type for Parallel Machines.* Technical Report TM 178, Computation Structures Group, MIT Lab. for Computer Science, Cambridge, MA 02139, September 1980.

[5] Henk Barendregt and Marc van Leeuwen. *Functional Programming and the Language TALE.* Technical Report TR 412, Mathematical Institute, Budapestlaan 6, 3508 TA Utrecht, The Netherlands, 1985.

[6] J. Driscoll, N. Sarnak, D. Sleator, and R. Tarjan. Making data structures persistent. In *Proc. 18th Annual ACM Symposium on Theory of Computing, Berkeley, CA*, pages 109–121, May 1986.

[7] Kim P. Gostelow and Robert E. Thomas. A view of dataflow. *AFIPS Conference Proceedings*, 48:629–636, June 1979.

[8] Paul Hudak. A semantic model of reference counting and its abstraction. In *Proc. 1986 ACM Conf. on Lisp and Functional Programming, MIT, Cambridge, MA*, pages 351–363, August 1986.

[9] Thomas Johnsson. Lambda lifting: transforming programs to recursive equations. In *Springer-Verlag LNCS 201 (Proc. Functional Programming Languages and Computer Architecture, Nancy, France)*, September 1985.

[10] Robert M. Keller. *FEL (Function Equation Language) Programmer's Guide.* Technical Report AMPS Technical Memorandum No. 7, University of Utah, Department of Computer Science, April 1983.

[11] David J. Kuck, R.H. Kuhn, D.A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *Proc. 8th ACM Symp. on Principles of Programming Languages*, pages 207–218, January 1981.

[12] Rishiyur S. Nikhil. *Id Nouveau Reference Manual.* Technical Report (Forthcoming), Computation Structures Group, MIT Lab. for Computer Science, Cambridge, MA 02139, 1987.

[13] Rishiyur S. Nikhil, Keshav Pingali, and Arvind. *Id Nouveau.* Technical Report CSG Memo 265, Computation Structures Group, MIT Lab. for Computer Science, Cambridge, MA 02139, July 1986.

[14] David A. Padua and Michael J. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12), December 1986.

[15] Kenneth R. Traub. *A Compiler for the MIT Tagged Token Dataflow Architecture.* Master's thesis, Technical Report TR-370, MIT Lab. for Computer Science, Cambridge, MA 02139, August 1986.

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

[17] Philip Wadler. A new array operation for functional languages. In *Proceedings of the Graph Reduction Workshop, Santa Fe, New Mexico, October 1986*, 1987.