I-Structures: Data Structures for Parallel Computing

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

Abstract

It is difficult to achieve elegance, efficiency and parallelism simultaneously 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, and show elegant, efficient and parallel solutions for the program examples in Id, a language with I-structures. The parallelism in Id is made precise by means of an operational semantics for Id as a parallel reduction system. I-structures make the language nonfunctional, but do not lose determinacy. Finally, we show that even in the context of purely functional languages, I-structures are invaluable for implementing functional data abstractions.

Categories and Subject Descriptors: D.3.2 [**Programming Languages**]: Language Classifications— *Applicative languages, Data-flow languages*; D.3.3 [**Programming Languages**]: Language Constructs— *Concurrent programming structures*; E.1 [**Data**]: Data Structures— *Arrays*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages— *Operational Semantics*

General Terms: Languages

Additional Key Words and Phrases: Functional Languages, Parallelism

[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

A preliminary version of this paper was published in *Proceedings of the Workshop on Graph Reduction, Santa Fe, New Mexico, USA, Springer-Verlag LNCS 279*, pages 336-369, September/October 1986.

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.

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). 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 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 also 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 [4, 25]. 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.

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

2 The Test Problems

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

2.1 Example A

Build a matrix with

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

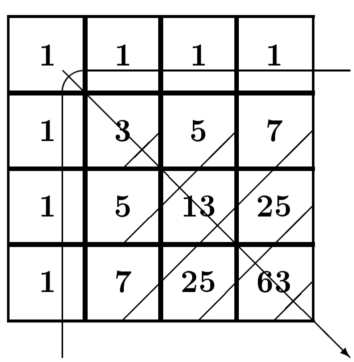
Note that the computation for each element is independent of all the others.

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 Henk Barendregt, and is illustrates the difficulties of dealing with computed indices. 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 subcomputation. And of course, in other applications the two types of sharing may be combined.

[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`".

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 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) ;
```

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 (m,n)
```

which returns an array whose index bounds are `(m,n)`, 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

[3] In Id, the comma is an infix tupling operation, so that the expression "$\mathtt{e}_1,\ldots,\mathtt{e}_n$" denotes a n-tuple whose components are the values of $\mathtt{e}_1$, ..., $\mathtt{e}_n$ respectively.

```
matrix ((m_i,n_i),(m_j,n_j))

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 [13] 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.

3.2.1 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:

```
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 <- 1 To m Do
      Next A = {For j <- 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 <- 1 To m Do
      Next A = {For j <- 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 [14], 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 [12]). 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

[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.

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]
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 [18, 22] 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:

```
B = {  B = array (1,n)
      In
        For i <- 1 To n Do
```

```
    Next B = update B A[i] i
  Finally B } ;
```

In order to parallelize the loop, the compiler needs to know something about the contents of `A`, such as 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).

[5]Originally, an I-structure was just a functional data structure with these two properties [7], and not a separate kind of object with its own operations.

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 [17, 8]. For example, the expression

```
make_array (m,n) f
```

where `(m,n)` is a pair (2-tuple) of integers and `f` is a function, returns an array whose index bounds are `(m,n)`, 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 (we hope) at significantly lower cost.

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

```
make_matrix ((m_i,n_i),(m_j,n_j)) f
```

produces a matrix where the `(i,j)`'th component contains `f (i,j)`. The first argument to `make_matrix` is a pair whose components are pairs of integers; it 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;

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 evaluated first and storage of the appropriate size allocated. A pointer to the array can now be returned immediately as the result of the `make_array` expression. Meanwhile, n independent processes are initiated, computing `(f 1)`, ..., `(f n)` respectively—each process, on completion, writes into the appropriate location in the array. Some synchronization mechanism is necessary at each array location so that a consumer that tries to read some `A[i]` while it is still empty is made to wait until the corresponding `(f i)` has completed. One way to achieve this synchronization is to use I-structure storage, where each location has "presence bits" to indicate whether the value is present or absent. I-structure storage is discussed in more detail in Section 5.

Another way to achieve this synchronization is by lazy evaluation: the bounds expression is evaluated first and storage of the appropriate size is allocated. Each location `A[i]` is then loaded with the suspension for `(f i)` and the pointer to the array is then returned. A subsequent attempt to read `A[i]` will force evaluation of the suspension, which is then overwritten by the value. In general, a fundamental activity of lazy evaluators— testing an expression to check if it is still a suspension— is really a synchronization test and also needs presence bits, although they are not usually referred to with that terminology.

This kind of nonstrictness 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 nonstrictness.

4.3 Example B (Wavefront)

A straightforward solution to the wavefront problem 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) ;

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] ;

g = f A ;
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`.

Note that in recurrences like this, it will be impossible in general to predict statically in what order the components must be filled to satisfy the dependencies, and so a compiler cannot always "preschedule" 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. This is true even for sequential implementations (lazy evaluation is one way to achieve this dynamic synchronization and scheduling).

Assuming the implementation handles such recurrences properly, the main inefficiency that remains is that the `If-Then-Else` is executed at every location. This problem arises even when there are no recurrences. In scientific codes, it is quite common to build a matrix with different filling functions for different regions, e.g., one function for boundary conditions and another for the interior. Even though this structure is known statically, `make_matrix` forces the use of a single filling function that, by means of a conditional, dynamically selects the appropriate function at each index. Compare this with the FORTRAN solution that would merely use separate loops to fill separate regions.

4.4 Example C (Inverse Permutation)

Unfortunately, `make_array` does not do so well on Example C. Recall that `B` contains a permutation of its indices, and we need to compute `A`, the inverse permutation.

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

Def g B i = find B i 1 ;

A = make_array (1,n) (g B) ;
```

The problem is that each `(g B i)` that is responsible for filling in 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 use a slightly different array primitive to address this problem. Consider

```
make_array_jv (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 g B i = B[i],i ;

A = make_array_jv (1,n) (g B) ;
```

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

[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 [28].

Note that this new primitive, make_array_jv, 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. Further, when make_array_jv is used for programs without index computations, such as Examples A and B, the compiler must now explicitly establish that the indices computed by the filling function form a legal permutation.

4.5 Example D (Shared Computation)

A straightforward attempt to solve the shared computation problem is:

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

A = make_array (1,n) fh ;
B = make_array (1,n) gh ;
```

This program, of course, does not share any computation— (h i) is repeated for each i for A and for B.

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

```
C = make_array (1,n) h ;

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

A = make_array (1,n) fh ;
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_array's 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 subexpression 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 constructor is a *complete* specification of a value, i.e., it includes the specification of the components. For example, "Cons e1 e2" specifies not only that the result is a cons-cell, but also specifies that its components are the values of e1 and e2.

For large data structures such as arrays, it is obviously not feasible in general to enumerate expressions for all the components as we do with `Cons`. Thus, their functional constructors must specify a regular way to *generate* the components. `Make_array` takes a "filling" parameter `f`, and it sets up n independent computations, with the i'th computation responsible for computing and filling the i'th location.

We saw three problems with this fixed control structure. The wavefront example showed that when the filling function is different for different regions of the array, they have to be selected dynamically using a conditional, even when the regions are known statically. In the inverse permutation problem, the fixed control struture was totally different from the desired control structure. Finally, there was no convenient way to express shared computation between the filling computations for two data structures.

The variant `make_array_jv` achieved some flexibility by leaving it up to each of the i computations to decide which index j it was responsible for. However, it still did not address the issue of shared computations, 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_jv` that uses association lists of index-and-value pairs together with his "listless transformer" [27], 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 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, we saw that the source of inefficiency is the fact that the various functional primitives impose too rigid a *control* structure on the computations 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 to create and manipulate I-structures, and in Section 5.2, we show how to code the programming examples using I-structures. In these sections, we rely on informal and intuitive explanations concerning parallelism and efficiency.

Finally, in Section 5.3 we make these explanations precise by presenting an operational semantics for a kernel language with I-structures, using a confluent set of rewrite rules. This section may be skipped on a first reading; however, there are several novel features about the rewrite rules, not usually found elsewhere in the functional languages literature. Even for the functional subset of Id, they capture precisely the idea of parallel, dataflow execution, which is parallel and normalizing; they describe precisely what computations are *shared*, an issue that is often left unspecified, and, finally, they are invaluable in developing one's intuitions about the read-write synchronization of parallel data structures, both functional and otherwise.

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 functional language with I-structures, we introduce three new constructs.

5.1.1 Allocation

An I-structure is allocated by the expression

```
I_array (m,n)
```

which allocates and returns an "empty" array whose index bounds are `(m,n)`. 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 multidimensional I-structures:

I_matrix $((m_i, n_i), (m_j, n_j))$

and so on.

5.1.2 Assignments and Constraints

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.

The assignment statement is only the simplest form of a constraint statement. A loop containing constraint statements and no "`Finally e`" clause is itself a constraint statement. A block with no "`In`" clause is a constraint statement. A procedure "`f`" may have a body that is a constraint statement; it is called using the constraint statement:

```
Call f x
```

In general, constraint statements appear intermixed with bindings in a block or loop-body.

5.1.3 Selection

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 nonempty, i.e., after some other part of the program has assigned the value.

There is no test for emptiness of an I-structure 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.

5.1.4 Discussion

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 store". 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".

An I-structure selection expression becomes an "I-fetch" request to the I-structure store. Every request is accompanied by a "tag", which can be viewed as the name of the continuation that expects the result. The controller for the I-structure store 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.

An I-structure assignment statement becomes an "I-store" request to the I-structure store. When such a request arrives the controller for the I-structure store checks the "empty" bit at that location. If it is empty, the value is stored there, the bit is toggled to "nonempty", 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 The Programming Examples

Let us now see how our programming examples are expressed in Id with I-structures.

5.2.1 Example A

The first example is straightforward:

```
{  A = I_matrix ((1,m),(1,n)) ;
   {For i <- 1 To m Do
      {For j <- 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 block until the value has been stored by the corresponding loop body.

5.2.2 Example B (Wavefront)

```
{  A = I_matrix ((1,m),(1,n)) ;
   {For i <- 1 To m Do
      A[i,1] = 1 } ;
   {For j <- 2 To n Do
      A[1,j] = 1 } ;
   {For i <- 2 To m Do
      {For j <- 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 for elements 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 <- m Downto 2 Do
   {For i <- 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.

5.2.3 Example C (Inverse Permutation)

```
{  A = I_array (1,n) ;
   {For i <- 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.

5.2.4 Example D (Shared Computation)

```
{  A = I_array (1,n) ;
   B = I_array (1,n) ;
   {For i <- 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.

5.3 Operational Semantics for a Kernel Language with I-structures

In this section we make the parallelism in the dataflow execution of Id more precise. First, some historical notes. For a long time, the parallelism of Id was described only in terms of dataflow graphs, the machine language of the dataflow machine. In [6], we made a preliminary attempt at describing it more abstractly, in terms of a set of rewrite rules. This was refined by Traub in [26], and subsequently by Ariola and Arvind in [3]. For a precise formalization and proofs of important properties such as confluence, the interested reader is referred to the last reference. Our description here borrows heavily from that reference, sacrificing much detail and omitting all proofs, in the interest of clarity.

The operational semantics are given as an *Abstract Reduction System*, i.e., a set of *terms* and a binary reduction relation that describes how to transform one term into another. The general form of a rewrite rule is:

$$E_1 \times ISS_1 \longrightarrow E_2 \times ISS_2$$

where $E \,\epsilon\, Expression$ and $ISS \,\epsilon\, \textit{I-Structure Stores}$. We begin by describing these syntactic categories.

5.3.1 Syntax of the Kernel Language, and its relation to Id

To simplify the exposition, we consider only a kernel language whose syntax is described in Figure 1. The translation of Id programs into the kernel language should be mostly self-evident; a few issues are discussed later in this section.

A program is a list of user-defined procedures and a main expression. The definitions may be recursive and mutually recursive.

The definitions in a *Program* are static, i.e., the definitions are not themselves part of the term being rewritten, though each definition represents an instance of the family of rewrite rules for the `apply` operator. The *Main* expression is only the initial expression in the term being rewritten. Expressions in the static part of the program, i.e., the right-hand sides of definitions and the main expression, are drawn only from *Initial Terms*. The syntactic category *Runtime Terms* is used to describe additional terms of interest and terms that may come into existence only during execution.

Each constant has an arity n (numerals, boolean constants, etc., are all considered to be constants of arity 0). A constant of arity $n > 0$ is always applied to n arguments, i.e., constants are never curried. Currying of user-defined procedures is simulated by the `apply` operator which is a constant of arity 2. For clarity, we will often omit the arity superscript, and we will sometimes use infix notation for well-known primitives; thus, "`x+y`" instead of "`plus x y`". We assume that procedure identifiers (*ProcIds*) are distinct from other identifiers.

In the kernel language, all subexpressions are named by identifiers. Thus, the Id expression "`23+34*45`" is written as follows in the kernel language:

```
{    x = plus² 23 y ;
     y = times² 34 45
In
     x }
```

Programs:

Program	::=	*Definition* ; $\cdots$; *Definition* ; *Main*	
Definition	::=	`Def` *ProcId* $\underbrace{\textit{Identifier} \cdots \textit{Identifier}}_{n}$ `=` *Expression*	$(n > 0)$

Initial Terms:

ProcId	::=	*Identifier*	
Main	::=	*Block*	
Expression	::=	*SimpleExpr*	
	|	$\textit{Constant}^n$ $\underbrace{\textit{SimpleExpr} \cdots \textit{SimpleExpr}}_{n}$	$(n > 0)$
	|	*Block*	
SimpleExpr	::=	$\textit{Constant}^0$ | *Identifier* | *ProcId*	
$\textit{Constant}^0$	::=	`0` | `1` | $\cdots$ | `true` | `false` | `nil` | $\cdots$	
$\textit{Constant}^1$	::=	`mk_closure` | `hd` | `tl` | $\cdots$	
$\textit{Constant}^2$	::=	`plus` | `minus` | `cons` |	
		`allocate` | `I_select` |	
		`apply` | $\cdots$	
$\textit{Constant}^3$	::=	`cond` | $\cdots$	
Block	::=	{ *Statement* ;	
		$\cdots$	
		Statement	
		`In`	
		SimpleExpr }	
Statement	::=	*Binding* | *Command*	
Binding	::=	*Identifier* `=` *Expression*	
Command	::=	`null` |	
		`I_set` *SimpleExpr* *SimpleExpr* *SimpleExpr*	

Runtime Terms:

$\textit{Constant}^1$	::=	$\texttt{I\_fetch}^1$	
Command	::=	`I_store` *Loc* *SimpleExpr*	
Value	::=	$\textit{Constant}^0$ | *Closure* | *I-structure*	
Closure	::=	`<closure` *ProcId* $\underbrace{\textit{Loc} \cdots \textit{Loc}}_{m}$ `>`	$(0 \leq m < n$, the # of args for *ProcId*)
I-structure	::=	`<I_structure` l u $\underbrace{\textit{Loc} \cdots \textit{Loc}}_{m}$ `>`	$(m = u - l + 1)$
Loc	::=	$\texttt{l}_0$ | $\texttt{l}_1$ | $\cdots$	
ISS	::=	`empty` |	
		`(`*Loc*`,` *Value*`)`,...,`(`*Loc*`,` *Value*`)` |	
		`inconsistent`	

Figure 1: Syntax of Kernel Language

Thus, kernel expressions are more tedious to write than their Id counterparts, but the rewrite rules are simplified considerably.

Blocks are like "letrec" blocks, i.e., the bindings may be recursive and mutually recursive; they follow the usual static scoping rules and the order of the bindings is not significant. For simplicity, bindings in blocks do not have formal parameters. There is no loss of expressive power— we assume that internal procedure definitions can be compiled out using techniques such as lambda-lifting [16].

In an Id conditional expression:

`if` e f g

nothing is executed in f or g until e is evaluated. Subsequently, exactly one of the expressions f and g is evaluated. Unfortunately, such contextual restrictions on rewrite rules usually complicate reasoning about a reduction system. Instead, we assume that the above conditional expression is first expressed in Id like this:

```
{ Def F x = f ;
  Def G x = g ;
  H = cond e F G
In
  H 0 }
```

where `cond` is a new primitive that simply returns its second or third argument depending on the boolean value of its first argument. The formal parameters `x` and the actual parameter `0` are just dummy arguments. This form can then be translated as usual to the kernel language. The same effect is achieved— exactly one of f and g is evaluated— without any contextual restrictions on the rewrite rules.

As a step towards translation into the kernel language, Id I-structure constructs are transformed as shown below.

`I_array` e_{bounds}	$\Longrightarrow$	`{ b =` e_{bounds} `;`
		`b`$_1$ `= I_select b 1 ;`
		`b`$_2$ `= I_select b 2 ;`
		`a = allocate b`$_1$ `b`$_2$
		`In`
		`a }`

e_a`[`e_i`]`	$\Longrightarrow$	`{ a =` e_a `;`
		`i =` e_i
		`In`
		`I_select a i }`

e_a`[`e_i`] =` e_v `;`	$\Longrightarrow$	`a =` e_a `;`
		`i =` e_i `;`
		`v =` e_v `;`
		`I_set a i v ;`

In Id, a block or a loop may be used as a constraint statement, in which case it does not return any value. It can always be converted into an expression returning a dummy value that is subsequently discarded by binding it to an unused identifier. Similarly, the statement `Call f x` can be converted into a binding `a = f x` where `a` is unused. Loop expressions can then be transformed to tail-recursive procedures in the standard way.

5.3.2 Runtime Expressions, Values and the I-structure store

Execution begins with the term:

$$Main \times \texttt{empty}$$

i.e., the main expression of the program (drawn from *Initial Terms*) and an empty I-structure store. As the reduction proceeds, the expression may be transformed to contain runtime terms, and the I-structure store grows with new location-value pairs. We use l and v (possibly with subscripts) as meta-variables for locations and values, respectively. The *Value* category corresponds exactly to "tokens" in the dataflow machine.

We assume an infinite supply of new locations. Location names are global, i.e., they do not follow scoping rules due to *Blocks*. The ordering of locations in the store is immaterial. In the rewrite rules, we use the notation:

$\texttt{ISS[(}l\texttt{,}v\texttt{)]}$

both as a pattern to match any store containing an $\texttt{(}l\texttt{,}v\texttt{)}$ pair and as a constructor to augment a store `ISS` with a new pair $\texttt{(}l\texttt{,}v\texttt{)}$.

5.3.3 Canonicalization

To simplify the exposition, we assume that rewrites are performed only on canonical terms. Thus, the overall reduction process can be viewed as repeatedly performing a rewrite followed by a canonicalization. There are three steps in canonicalization.

Block Flattening

Nested blocks can be flattened, whether the nested block is in the bindings or in the "`In`" expression:

$$
\boxed{\begin{array}{lllll}
\{ & B_1 \;; & & & \\
 & B_2 \;; & & & \\
 & \cdots & & & \\
 & \texttt{x} = & \{ & C_1 \;; & \\
 & & & C_2 \;; & \\
 & & & \cdots & \\
 & & \texttt{In} & & \\
 & & & R \; \} & \\
 & \cdots & & & \\
 & B_n & & & \\
\texttt{In} & & & & \\
 & \{ & D_1 \;; \quad \} & & \\
 & & D_2 \;; & & \\
 & & \cdots & & \\
 & \texttt{In} & & & \\
 & & S \; \} & &
\end{array}}
\longrightarrow
\boxed{\begin{array}{ll}
\{ & B_1 \;; \\
 & B_2 \;; \\
 & \cdots \\
 & C'_1 \;; \\
 & C'_2 \;; \\
 & \cdots \\
 & \texttt{x} = R' \;; \\
 & \cdots \\
 & B_n \;; \\
 & D'_1 \;; \\
 & D'_2 \;; \\
 & \cdots \\
\texttt{In} & \\
 & S' \; \}
\end{array}}
$$

The "primed" terms on the right indicate suitable α-renaming of identifiers to avoid name clashes.[7] As will become evident later, an inner block may have zero bindings to be lifted into the outer block.

Identifier Substitution and Binding Erasure

The substitution rules for identifiers are designed very carefully to model the sharing of computations precisely:

1. If x and y are distinct identifiers and there is a binding "x = y" in a term, then we can eliminate this binding from the term after replacing all uses of x with y.
2. If x is an identifier and v a value, and there is a binding "x = v" in a term, then we can eliminate this binding from the term after replacing all uses of x with v.

Of course, in the first rule, one must be cognizant of scoping due to blocks to avoid the inadvertant capture of y.

The major deviation from substitution rules for the lambda calculus, say, is that x must be bound to a *Value* and not to an arbitrary expression before it can be substituted. This ensures that unevaluated expressions are never duplicated, i.e., they are evaluated exactly once.

`Null` erasure

All `null` statements in blocks can be eliminated.

5.3.4 Rewrite Rules

The general notation for a rewrite rule is:

$$\boxed{M[p_1] \times ISS_1 \longrightarrow M[p_2] \times ISS_2}$$

where *M[p]* is the main program expression and p is a pattern that identifies any matching subexpression e, and *ISS* is the I-structure store. The subexpression e is called a *redex*, or reducible expression.

For most rewrite rules, the I-structure store is irrelevant; these rules are written more succinctly:

$$\boxed{p_1 \longrightarrow p_2}$$

[7] However, recall that *location* names are not identifiers, so they are never renamed.

with the understanding that they really abbreviate the full form above.

There may be more than one redex in the program term. The dataflow rule requires all such redexes to be reduced simultaneously. Of course, in a real implementation only a subset of redexes can be rewritten at any step. We discuss this issue in more detail in Section 5.3.7.

δ rules:

We assume suitable δ rules for all the primitive functions. For example:

$$\boxed{\begin{array}{lcl} \underline{m}+\underline{n} & \longrightarrow & \underline{m+n} \\ \vdots & & \vdots \end{array}} \qquad (m \text{ and } n \text{ numerals})$$

Conditionals:

$$\boxed{\begin{array}{lcl} \texttt{cond true } f\ g & \longrightarrow & f \\ \texttt{cond false } f\ g & \longrightarrow & g \end{array}}$$

User-defined procedures: Closures, Partial and Full Applications:

Suppose we had the following user-defined procedure of n-arguments:

$\texttt{Def}\ f\ x_1\ \cdots\ x_n\ \texttt{=}\ E_{body}\ \texttt{;}$

Initially, f appears in the program as an argument to `mk_closure`, whose rewrite rule is shown below:

$$\boxed{\texttt{mk\_closure}\ f \longrightarrow \texttt{<closure}\ f\texttt{>}} \qquad (f \in \mathit{ProcId})$$

The closure is then applied to arguments, one at a time. However, unless the argument is its "last" argument, we simply create a new closure, allocating a new location to hold the value of the argument:

$$\boxed{\begin{array}{lcll} \texttt{apply <closure}\ f\ l_1\ \cdots\ l_{j-1}\texttt{>}\ e_j & \longrightarrow & \{ & \texttt{I\_store}\ l_j\ e_j \\ & & \texttt{In} & \\ & & & \texttt{<closure}\ f\ l_1\ \cdots\ l_{j-1}\ l_j\texttt{>}\ \} \end{array}} \qquad \begin{array}{l} j < n, \\ \text{new } l_j \end{array}$$

Note, however, that the new closure is ready to be used as a value *immediately*, even if the argument e_j is not yet a value. For example, the closure may now be applied to more arguments. However, the rules ensure that e_j is evaluated exactly once.

It is only when a closure is applied to the "last" argument expression that we invoke the procedure body, i.e., rewrite using the procedure definition:

$$\boxed{\begin{array}{lcll} \texttt{apply <closure}\ f\ l_1\ \cdots\ l_{n-1}\texttt{>}\ e_n & \longrightarrow & \{ & x_1\ \texttt{=}\ \texttt{I\_fetch}\ l_1\ \texttt{;} \\ & & & \cdots \\ & & & x_{n-1}\ \texttt{=}\ \texttt{I\_fetch}\ l_{n-1}\ \texttt{;} \\ & & & x_n\ \texttt{=}\ \texttt{e}_n \\ & & \texttt{In} & \\ & & & E_{body}\ \} \end{array}}$$

Again, note the parallel dataflow behavior. The body of the procedure becomes ready for evaluation even though all the argument expressions may still be unevaluated. However, the substitution rules for identifiers and for `I_fetch` ensure that the arguments are not substituted into the body until they have reduced to values. Further, each argument is evaluated exactly once, even if it is used many times in the procedure.

I-structure Operations

Allocation:

$$\boxed{\texttt{allocate}\ \underline{m}\ \underline{n}\ \longrightarrow\ \texttt{<I\_structure}\ m\ n\ l_m\ l_{m+1}\ \cdots\ l_{n-1}\ l_n\texttt{>}} \qquad \begin{array}{l} m\texttt{,}\ n\ \texttt{integer values,} \\ m \leq n\texttt{,} \\ l_m \cdots l_n\ \texttt{new} \end{array}$$

`I_set`'s and `I_select`'s, after address computations, become `I_fetch`'es and `I_store`'s against specific locations:

$$\boxed{\texttt{I\_set <I\_structure}\ m\ n\ l_m\ \cdots\ l_i\ \cdots\ l_n\texttt{>}\ \underline{i}\ e\ \longrightarrow\ \texttt{I\_store}\ l_i\ e} \qquad \begin{array}{l} \underline{i}\ \text{a value,} \\ m \leq \underline{i} \leq n \end{array}$$

$$\boxed{\texttt{I\_select <I\_structure}\ m\ n\ l_m\ \cdots\ l_i\ \cdots\ l_n\texttt{>}\ \underline{i}\ \longrightarrow\ \texttt{I\_fetch}\ l_i} \qquad \begin{array}{l} \underline{i}\ \text{a value,} \\ m \leq \underline{i} \leq n \end{array}$$

Note that the I-structure and index arguments must be values, but the third argument (e) to `I_set` need not be a value. There are various ways to handle out-of-bounds errors, but we do not address them here.

An `I_store` augments the I-structure store with a new location-value pair, provided that the store does not already contain the location. If it does already contain the location, the *entire* I-structure store goes to an inconsistent state:

$$\boxed{\texttt{M[I\_store}\ l\ v\texttt{]}\ \times\ \texttt{ISS}\ \longrightarrow\ \texttt{M[null]}\ \times\ \texttt{ISS[(}l\texttt{,}v\texttt{)]}} \qquad \nexists\ \texttt{any (}l\texttt{,}v'\texttt{) in ISS}$$

$$\boxed{\texttt{M[I\_store}\ l\ v\texttt{]}\ \times\ \texttt{ISS[(}l\texttt{,}v'\texttt{)]}\ \longrightarrow\ \texttt{M[null]}\ \times\ \texttt{inconsistent}}$$

An `I_fetch` against a location can be reduced only after a value is present in that location:

$$\boxed{\texttt{M[I\_fetch}\ l\texttt{]}\ \times\ \texttt{ISS[(}l\texttt{,}v\texttt{)]}\ \longrightarrow\ \texttt{M[}v\texttt{]}\ \times\ \texttt{ISS[(}l\texttt{,}v\texttt{)]}}$$

Functional Data Structure Operations

These can be expressed in terms of I-structure operations. Construction:

```
cons e1 e2  -->  {    c = allocate 1 2 ;
                      I_set c 1 e1 ;
                      I_set c 2 e2
                 In
                      c }
```

and selection:

```
hd e  ⟶  I_select e 1
tl e  ⟶  I_select e 2
```

We include these as rewrite rules only because it then allows us to define a very simple syntactic criterion to limit user-programs to functional programs, i.e., by omitting *Commands* from Initial Terms while keeping them in Runtime Terms. But for this reason, we could treat the above rewrite rules as ordinary definitions supplied by the user, or as a compile-time transformation.

5.3.5 An Example

Let us look at a small example that demonstrates the nonstrict behavior of data structures. The following Id expression:

```
{  p = cons 23 p
In
   hd p }
```

defines p to be an “infinite” list of 23’s, and returns only the first element of it. Below, we show a possible reduction. At each step, we enclose the chosen redex(es) in a box. Many steps, especially canonicalization steps, are omitted for brevity:

```
        { p = cons 23 p ;
          r = hd p
        In
          r } × empty
⟶
        { c = allocate 1 2 ;
          I_set c 1 23 ;
          I_set c 2 p ;
          p = c ;
          r = I_select p 1
        In
          r } × empty
⟶
        { I_set <I-structure 1 2 L1 L2> 1 23 ;
          I_set <I-structure 1 2 L1 L2> 2 <I-structure 1 2 L1 L2> ;
          r = I_select <I-structure 1 2 L1 L2> 1
        In
          r } × empty
        { I_store L1 23 ;
          I_store L2 <I-structure 1 2 L1 L2> ;
          r = I_fetch L1
        In
          r } × empty
```

$\longrightarrow$

```
{ r = [I_fetch L1]
In
  r } × (L1,23)(L2,<I-structure 1 2 L1 L2>)
```

$\longrightarrow$

```
{
In
  23} × (L1,23),(L2,<I-structure 1 2 L1 L2>)
```

In the final state, there are no remaining statements in the block, and the result value is 23. The final I-structure store contains a self-referential structure.

5.3.6 Termination and Answers

The reduction terminates as soon as we reach a state $N \times$ *ISS* where either *ISS* $=$ `inconsistent` or N is in *normal form*, i.e., it contains no redexes. It is useful to identify the following four categories of termination:

- Proper termination with answer v:

  `{ In` v `}` $\times$ *ISS*

  where $v \in$ *Value*, *ISS* $\neq$ `inconsistent`.

- Improper termination with inconsistent I-structure store:

  N $\times$ `inconsistent`

  Note that N does not have to be in normal form. Further,

  $\forall\ M, N$ $\quad M$ $\times$ `inconsistent` $=$ N $\times$ `inconsistent`

- Improper termination with answer v:

  `{` s_1 `; ... ;` s_k `In` v `}` $\times$ *ISS*

  where $k \geq 1$, $v \in$ *Value*, *ISS* $\neq$ `inconsistent`.

- Improper termination with deadlock:

  `{` s_1 `; ... ;` s_k `In` e `}` $\times$ *ISS*

  where $k \geq 1$, $e \not\in$ *Value*, *ISS* $\neq$ `inconsistent`.

The last two situations arise only if there are unreduced `I_fetch` expressions in the normal form. Here are three pathological Id programs with these behaviors, respectively:

```
{ a = I_array (1,1);
  x = a[1]
In
  23 }
```

```
{ a = I_array (1,2);
  a[1] = a[2] ;
  a[2] = a[1] ;
In
  a }
```

```
{ x = y+1;
  y = x+1;
In
  23 }
```

In our dataflow implementation of Id, we print an answer as soon as it is available. When the program subsequently terminates, additional information about improper termination is printed, if necessary.

5.3.7 Discussion of Kernel Language and Rewrite rules

Reduction Strategies

The rewrite rules do not themselves specify any strategy for choosing redexes. Our implementation uses a "parallel dataflow strategy", i.e., it attempts to evaluate all redexes concurrently. Of course, because of limited resources, only some subset of available redexes will be reduced at each step. A poor choice of redexes may place severe demands on the resources of the machine. For example, an array may be allocated long before it is actually used, thus utilizing storage inefficiently. We have studied various approaches to controlling the unfolding of a parallel computation so that it uses resources efficiently [9, 11, 10]. These issues are relevant to all parallel languages, but the details are beyond the scope of this paper.

In our rewrite rules, no redex is ever duplicated or discarded. Thus, the number of reductions performed is independent of the reduction strategy. As a consequence, the only expressions in an Id program that are not evaluated are those that are not selected by a conditional. Thus, consider the following two Id programs:

```
{ a = I_array (1,1) ;



  z = if p
        then { a[1] = 23 In a[1] }
        else { a[1] = 24 In a[1] }
In
  z }
```

```
{ a = I_array (1,1) ;
  x = { a[1] = 23 In a[1] } ;
  y = { a[1] = 24 In a[2] } ;
  z = if p
         then x
         else y
In
  z }
```

The program on the left will terminate properly with value 23 or 24; only one of the two array assignments will be performed. The program on the right will always terminate improperly with an inconsistent store because both array assignments will always be performed. In our experience, this has never been a complication— it is not much different from the way in which we control unbounded recursion in ordinary programming languages using conditionals.

The functional subset and its relation to lazy languages

If we wish to limit ourselves to purely functional programs, the following syntactic criterion is sufficient: there should be no *Commands* in the initial program. It is also reasonable to disallow `allocate` expressions as they are quite useless without *Commands*. Note that *Commands* may still appear during the execution of `cons` and partial applications; we disallow them only in initial programs. However, it is easy to prove that under these conditions, it is impossible for two `I_stores` to go to the same location and, therefore, it is impossible to produce an `inconsistent` store.

Let us reexamine the issue of termination for the functional subset. Consider the following Id program:

```
Def diverge x = diverge x ;
Def f y = 5 ;

f (diverge 1)
```

This program will not terminate under our rewrite rules because we never discard "unnecessary" expressions like "`diverge 1`". The reason is that in general, with I-structures, it may not be safe to do so— any expression may ultimately make the store inconsistent, thus changing the answer. For confluence, therefore, we cannot discard any expression.

If we restrict ourselves to functional programs, however, this danger of inconsistency cannot arise, and so it is safe to add the following rewrite rule (let us call it the *discard* rule):

```
{    s1 ;   -->  {
     ...         In
     sn ;             v }
In
     v }
```

$v \in$ *Values*

Note that this is the *only* rule that can discard a redex. Further, no rule ever duplicates a redex. Thus, without the discard rule, the number of reductions performed (and, thus, the termination behavior) is independent of the reduction strategy, so that a normal-order strategy would not be particularly useful. With the discard rule, however, the reduction strategy does affect the number of reductions performed and can affect termination. Because our rewrite rules also capture the sharing of expressions, they would, under normal-order, accurately describe lazy evaluation and graph-reduction machines [23].

Note, however, that with the discard rule, the parallel dataflow strategy will produce exactly the same termination behavior that normal-order would, even though it may perform some extra, unnecessary reductions. Thus functional Id, even though it does not use lazy evaluation, implements exactly the same *nonstrict* semantics that normal-order does, i.e., functions can return values even if their arguments do not terminate. To our knowledge, Id is unique in this respect— every other functional language that implements nonstrict semantics does so using lazy evaluation.[8]

Confluence

The confluence of the reduction system has been proved in [3]. Note that confluence holds for the entire kernel language, including all I-structure operations, and not just for a functional subset.

[8]Perhaps for this reason, nonstrictness is often incorrectly equated with laziness in the literature.

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 (m,n) f = {  A = array (m,n) ;
                            {For i <- m To n 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.

Similarly, here is an efficient, parallel implementation for make_matrix:

```
Def make_matrix ((mi,ni),(mj,nj)) f =
                         {  A = matrix ((mi,ni),(mj,nj)) ;
                            {For i <- mi To ni Do
                              {For j <- mj To nj Do
                                 A[i,j] = f (i,j) }}
                          In
                            A } ;
```

A functional vector sum:

```
Def map2 f A B = { m,n = bounds A ;
                   C = array (m,n) ;
                   {For i <- m To n Do
                      C[i] = f A[i] B[i] }
                 In
                   C } ;

vector_sum = map2 (+) ;
```

Here, we first define a more general abstraction map2 for applying a binary function f to each pair of elements taken itemwise from two vectors, and then define vector_sum as the partial application of map2 to the specific binary function "+". 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.

As another demonstration of the usefulness of programming with abstractions like map2, consider a function to add two vectors of vectors (i.e., a vector sum where the components are not numbers, but vectors themselves):

```
vector_of_vectors_sum = map2 vector_sum ;
```

An implementation of the functional make_array_jv primitive:

```
Def make_array_jv (m,n) f = {  A = array (m,n) ;
                               {For i <- 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 (m,n) f = {  A = array (m,n) ;
                                 B = array (m,n) ;
                                 {For i <- m To n 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 it is straightforward for the programmer to use I-structures to implement any desired functional array abstractions— the solutions are perspicuous, efficient, and there is no loss of parallelism.

It is likely that even if abstractions like make_array are supplied as primitives, I-structures are a useful implementation mechanism for the compiler. Supplying such abstractions as primitives is useful for another reason. Consider an abstraction such as make_array_jv which, if given an improper filling function f, could cause multiple I_stores against the same location. Currently, the effect of this is drastic— the entire program immediately terminates improperly with an inconsistent store. With functional abstractions as primitives, it is possible to localize such errors. The implementation of make_array_jv could examine all the indices computed by f before releasing the array pointer to its caller. If an index was computed twice, the array value returned could be an error value, without causing the whole program to blow up. Such an implementation comes with the loss of some, but not all, concurrency, i.e., all the indices, but not the corresponding values, need to be computed before the array is returned.

Such localization is not possible in a language with I-structures because, unlike functional constructors, the index calculations may be spread over arbitrary regions of the program.

6.2 Nonfunctional Array Abstractions

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

```
Def fill A ((mi,ni),(mj,nj)) f =
                     {For i <- mi To ni Do
                        {For j <- mj To nj 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 ((mi,ni),j) f = {For i <- mi To ni Do
                                   A[i,j] = f (i,j)} ;

Def fill_row  A (i,(mj,nj)) f = {For j <- mj To nj 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 for 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

$$a_0 + a_1 x + a_2 x^2 + a_3 x^3 ... + a_n x^n$$

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 first need 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 mention some solutions to this problem at the end of the next section.[9] (See also [24] for a connection between accumulators and logic variables.)

[9] In [28], 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.

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.

I-structures grew out of a long-standing goal in our group to have functional languages suitable for *general-purpose* computation, which included scientific computations and the array data-structures that are endemic to them. A historical perspective: the term "I-structure" has been used for two separate concepts. One is an architectural idea, i.e., a particular means of implementing a synchronization mechanism in hardware [5]. The other is a language construct, a way to express incrementally-constructed data structures. The two are independent— the architectural support makes sense even for FORTRAN, and the language constructs make sense even on stock hardware. The emphasis in this paper is on the language construct.

Originally ([7], 1981), an "I-structure" was not a separate kind of object with its own operations; rather, a *functional* array built using a fine-grained `update`-like operator in a particular incremental manner (with no repeated indices) was termed an I-structure. It was hoped that the compiler, through analysis, would be able to recognize such incremental constructions and to implement them efficiently using destructive updates. This approach was later abandoned after it was judged to be infeasible.

The connection with logic variables was originally inspired by Lindstrom's FGL+LV [19] in 1985/1986. This clarification of the role of I-structure cells gave us the basis on which to incorporate the current view of I-structures into the language as first class objects with their own operations and semantics [21, 6], which is substantially different from the original view of I-structures. Further progress on the semantics of logic variables in functional languages is reported in [15].

The introduction of any nonfunctional feature (such as I-structures) into a functional language is not without cost— the language loses referential transparency and with it, 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. For example, values bound in these two statements are not semantically equivalent:

```
AA = { a = I_array (1,10)
     In
       a, a } ;

BB = I_array (1,10), I_array (1,10)
```

They can be distinguished by the following function:

```
Def f XX = { Xa,Xb = XX;
             Xa[1] = 23;
             Xb[1] = 24
           In
             25} ;
```

When applied to AA, f will produce an inconsistent store, whereas when applied to BB, it terminates properly with value 25. 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 "I_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) = (m,n)
```

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

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

```
Def alloc (m,n) = {  array_bounds (x) = (m,n)
                     In x } ;
```

To prevent this, procedural 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 (m,n) x = { array_bounds (a) = m,n ;
                            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 (as shown in Section 6), 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 techniques available to reason about, and to manipulate functional programs.

Postscript:

Writing scientific applications in Id has always been part of our methodology to evaluate existing Id constructs and to suggest new ones. Based on the substantial experience we have had with I-structures since we began writing this paper in 1986, we have recently devised a new, *functional* notation for arrays called "array comprehensions" [20] that can, in fact, be used to express all four examples used in this paper. For example, the wavefront program:

```
{  A = {matrix (1,m),(1,n)
       | [1,1] = 1
       | [i,1] = 1             || i <- 2 To m
       | [1,j] = 1             || j <- 2 To n
       | [i,j] = A[i-1, j ] +
                 A[i-1,j-1] +
                 A[ i ,j-1]    || i <- 2 To m & j <- 2 To n}
In
   A }
```

An array comprehension begins with a specification of the shape (e.g., `matrix`) and size (index bounds) of the data structure, and contains one or more *region-specification* clauses. For example, the clause above specifies that the contents of location (1,1) is 1; the second clause specifies that the contents of location (i,1) is 1 for all $2 \leq i \leq m$; and so on. This program essentially compiles into exactly the program shown in Section 5.2, i.e., with the same parallelism and space efficiency.

Another example: the inverse permutation program:

```
{array (1,n)
| [B[i]] = i || i <- 1 to n}
```

A simple extension to array comprehensions also handles some "accumulator" problems. A 10-bucket histogram of a million samples:

```
{array (1,10)
| [i] = 0 || i <- 1 to 10
accumulate (+)
| [classify s] gets 1 || s <- million_samples }
```

The clause before `accumulate` specifies the initial value of the histogram buckets (all zeroes). The `accumulate` clause specifies that "+" is the accumulation operator. The final clause specifies that for each sample `s`, 1 is added to the j'th bucket, where `s` is classified into the j'th bucket.

The array comprehension construct greatly enlarges the set of applications that can be captured succinctly in a purely functional style without losing parallelism and efficiency. However, we frequently encounter other applications that still require the greater flexibility of I-structures. Work on generalizing array comprehensions continues.

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] W. 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. 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] Z. Ariola and Arvind. P-TAC: A Parallel Intermediate Language. Technical report, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, USA, January 1989.

[4] Arvind and D. E. Culler. Dataflow Architectures. In *Annual Reviews in Computer Science*, volume 1, pages 225–253. Annual Reviews Inc., Palo Alto, CA, 1986.

[5] Arvind and R. S. Nikhil. Executing a Program on the MIT Tagged-Token Dataflow Architecture. *IEEE Transactions on Computers*, 1989 (to appear). An earlier version appeared in *Proceedings of the PARLE Conference, Eindhoven, The Netherlands*, Springer-Verlag LNCS 259, June 15-19, 1987.

[6] Arvind, R. S. Nikhil, and K. K. Pingali. Id Nouveau Reference Manual, Part II: Semantics. Technical report, Computation Structures Group, MIT Lab. for Computer Science, 545 Technology Square, Cambridge, MA 02139, USA, April 1987.

[7] Arvind and R. E. Thomas. I-Structures: An Efficient Data Structure for Functional Languages. Technical Report MIT/LCS/TM-178, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, USA, October 1981.

[8] H. Barendregt and M. van Leeuwen. Functional Programming and the Language TALE. Technical Report TR 412, Mathematical Institute, Budapestlaan 6, 3508 TA Utrecht, The Netherlands, 1985.

[9] D. E. Culler. Resource Management for the Tagged Token Dataflow Architecture. Technical Report TR-332, MIT Lab. for Computer Science, 545 Technology Square, Cambridge, MA 02139, 1985.

[10] D. E. Culler. *Effective Dataflow Execution of Scientific Applications*. PhD thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, 1989 (expected).

[11] D. E. Culler and Arvind. Resource Requirements of Dataflow Programs. In *Proceedings of the Fifteenth Annual International Symposium on Computer Architecture, Honolulu, Hawaii*, May 1988.

[12] J. Driscoll, N. Sarnak, D. Sleator, and R. Tarjan. Making Data Structures Persistent. In *Proceedings of the 18th Annual ACM Symposium on Theory of Computing, Berkeley, CA*, pages 109–121, May 1986.

[13] K. P. Gostelow and R. E. Thomas. A View of Dataflow. *AFIPS Conference Proceedings*, 48:629–636, June 1979.

[14] P. Hudak. A Semantic Model of Reference Counting and its Abstraction. In *Proceedings of the 1986 ACM Conf. on Lisp and Functional Programming, MIT, Cambridge, MA*, pages 351–363, August 1986.

[15] R. Jagadeesan, P. Panangaden, and K. K. Pingali. A Fully Abstract Semantics for a Functional Language with Logic Variables. In *Proceedings of the Fourth IEEE Symposium on Logic in Computer Science, Asilomar, CA*, June 5-8 1989.

[16] T. Johnsson. Lambda Lifting: Transforming Programs to Recursive Equations. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture, Nancy, France, Springer-Verlag LNCS 201*, September 1985.

[17] R. 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.

[18] D. J. Kuck, R. Kuhn, D. Padua, B. Leasure, and M. Wolfe. Dependence Graphs and Compiler Optimizations. In *Proceedings of the 8th Annual ACM Symposium on Principles of Programming Languages*, pages 207–218, January 1981.

[19] G. Lindstrom. Functional Programming and the Logic Variable. In *Proceedings of the 12th Annual ACM Symposium on Principles of Programming Languages*, pages 266–280, 1985.

[20] R. S. Nikhil. Id (Version 88.1) Reference Manual. Technical Report CSG Memo 284, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, August 1988.

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

[22] D. A. Padua and M. J. Wolfe. Advanced Compiler Optimizations for Supercomputers. *Communications of the ACM*, 29(12), December 1986.

[23] S. L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987.

[24] K. K. Pingali and K. Ekanadham. Accumulators: New Logic Variable Abstractions for Functional Languages. In *Proceedings of the Conference on Foundations of Software Technology and Theoretical Computer Science, Springer-Verlag LNCS 338*, pages 377–399, December 1988.

[25] K. 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.

[26] K. R. Traub. Sequential Implementation of Lenient Programming Languages. Technical Report TR-417, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, May 1988. Ph.D. thesis.

[27] P. Wadler. Listlessness is Better than Laziness: Lazy Evaluation and Garbage Collection at Compile Time. In *Proceedings of the 1984 ACM Conferernce on Lisp and Functional Programming, Austin, TX*, pages 45–52, August 1984.

[28] P. Wadler. A New Array Operation for Functional Languages. In *Proceedings of the Workshop on Graph Reduction, Santa Fe, New Mexico, USA, Springer-Verlag LNCS 279*, pages 328–335, September 1986.

Contents