LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

M-Structures:
Extending a Parallel, Non-strict, Functional Language with State

Computation Structures Group Memo 327
March 18, 1991

Paul S. Barth
Rishiyur S. Nikhil
Arvind

In *Proceedings on Functional Programming and Computer Architecture*,
Cambridge, MA, August 28-30, 1991.

This report describes research done at the Laboratory for Computer Science of the Massachusetts Institute of Technology. Funding for the Laboratory is provided in part by the Advanced Research Projects Agency of the Department of Defense under the Office of Naval Research contract N00014-89-J-1988.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

M-Structures: Extending a Parallel, Non-strict, Functional Language with State

Paul S. Barth, Rishiyur S. Nikhil and Arvind
Massachusetts Institute of Technology

Laboratory for Computer Science
545 Technology Square, Cambridge, MA 02139, USA
{barth,nikhil,arvind}@au-bon-pain.lcs.mit.edu

Abstract

Functional programs are widely appreciated for being "declarative" (algorithms are not *over-specified*) and implicitly parallel. However, when modeling state, both properties are compromised because the state variables must be "threaded" through most functions. Further, state "updates" may require much copying. A solution is to introduce assignments, as in ML and Scheme; however, for meaningful semantics, they resort to strict, sequential evaluation.

We present an alternative approach called *M-structures*, which are imperative data structures with implicit synchronization. M-structures are added to Id, a parallel non-strict functional language. We argue that some programs with M-structures improve on their functional counterparts in three ways: they are (1) more *declarative*; (2) more *parallel*; and (3) more *storage efficient*. Points (1) and (2) contradict conventional wisdom. We study two problems: histogramming a tree, and graph traversal, comparing functional and M-structure solutions.

1 Introduction

Functional programming is widely appreciated for its declarative nature. By "declarative" we mean that one does not have to *over-specify* the details of an algorithm. This is sometimes paraphrased as: "Say *what* you want done, not *how* you want it done".

However, some programs cannot be expressed naturally in a purely functional style. For example, a program to assign unique identifiers to the nodes in a tree must thread the supply of unique identifiers through the traversal of the tree; this "plumbing" can be quite messy. Another weakness of functional languages is that they are difficult to implement efficiently— there is too much copying of data structures.

A common approach to addressing these weaknesses is to extend a functional language with state. Both ML and Scheme are examples of this. However, in order that these imperative features have a meaningful semantics, the language is usually made into a strict, call-by-value, sequential language. For functional programs, this is an over-specification of the order of evaluation, and may thus be viewed as a loss of declarativeness.

Another issue is parallelism. In pure functional languages, parallelism is implicit. Annotations for parallelism may be used, but these can be viewed as hints and do not add to the intellectual burden of the programmer. However, in sequential functional languages with imperative extensions, parallelism is usually re-introduced through another layer of constructs such as threads, forks, semaphores, futures, *etc.* These features usually make the language more complex. Further, when two such parallel programs are composed, their parallelism does not compose, because of the underlying strict semantics. The difference in resulting parallelism can be exponential, as demonstrated in [14].

Parallelism can also be curtailed by the threading alluded to above. Threading ensures determinacy at the expense of serializing access to the shared data structure. For a large class of algorithms, determinacy can be ensured in the presence of asynchronous updates to a shared structure. For example, accumulation and set operations can use associative and commutative properties to produce determinate results under any sequence of operations. For such applications, only the *atomicity* of each operation need be guaranteed to ensure determinate results. Again, the threading required by functional programming over-specifies, by serializing, an inherently parallel algorithm.

Let us call the traditional approaches A and B:

> **Approach A:** purely functional (strict or non-strict) with implicit parallelism or annotations for parallelism.
>
> **Approach B:** strict functional + sequential evaluation + imperative extensions + threads and semaphores for parallelism.

In the literature, the landscape of functional programming techniques is normally defined by these two approaches. In this paper we would like to open up an entirely new frontier:

> **Approach C:** non-strict functional + implicitly parallel evaluation + M-structures + occasional sequentialization.

M-structures are mutable data structures manipulated with implicitly synchronized imperatives called *take* and *put*, which are quite different from the traditional assignment statement.

The main points we would like to demonstrate about Approach C are:

(1) Programs in Approach C are often more declarative than their functional counterparts, because they are less specific about the order in which things should be computed. By omitting this level of detail, programs in Approach C are often shorter and easier to read than the pure functional versions.

(2) Programs in Approach C often have much more parallelism than their functional counterparts.

(3) Programs in Approach C are often much more efficient than their functional counterparts in terms of heap storage used and total number of instructions executed.

The first two points come to us as a great surprise, because they contradict our previous intuitions (and, we believe, conventional wisdom)— we always took it for granted that functional programs are more declarative than programs with state, and that functional programs always had more parallelism than programs with state. See [3] for studies of implicit parallelism in functional Id programs.

In fact, point (2) is a very curious reversal of conventional wisdom. Whereas we always thought that programs with state needed powerful analysis (*i.e.*, dependence analysis) in order to recover the parallelism that is already evident in functional programs, we find here that it is the functional programs that would need powerful analysis to recover the parallelism that is evident in Approach C. To our knowledge, no such analysis techniques exist yet.

The final point, storage efficiency, is also an issue in sequential functional programs. Many analysis techniques have been developed which attempt to improve storage efficiency through reuse, such as abstract reference counting[10] and single-threadedness analysis. Unfortunately, most published techniques seem to be based on an underlying sequential, strict semantics, sometimes restricted to first-order languages and flat domains for data structures. Thus, non-strict, implicitly parallel languages are unlikely to benefit from these techniques.

We hasten to add that Approach C is not without its problems! As soon as one adds state, one loses referential transparency (RT), and any benefits that may go with it. For example, RT provides a foundation for equational reasoning, which may be used during compilation for common subexpression elimination, proofs of invariants used in loop optimization, *etc.* Equational reasoning may also be used to prove functional programs correct. Our intent is not to challenge this work, but rather to demonstrate some programs which benefit, in both declarativeness and efficiency, from violating RT. Such benefits have motivated other "functional" languages, such as ML and Scheme, to introduce imperative constructs. These languages, which comprise Approach B, are still considered to be functional languages because the style of programming in these languages is "mostly functional". We suggest the same for Approach C—programs are still written in a mostly functional style. Further, we claim that the non-strictness and implicit parallelism of Approach C yields a more functional programming style than Approach B with explicit parallelism constructs. Regardless, Approaches B and C will require the development of new foundations for proving correctness, such as proposed by [6].

In this paper, we will demonstrate our claims by comparing Approach A (functional programs) and Approach C (M-structures) for two example applications:

computing a histogram of values, and performing a graph traversal. The declarativeness of these programs is critiqued, and their performance measured on a parallel simulator. The rest of this paper is organized as follows: Section 2 describes the core functional language, its non-strict semantics and parallel execution model, the M-structure extensions, and our parallel simulator. Sections 3 and 4 explore the two example programs, respectively, in some detail. Section 5 discusses our results, and outlines a method for reasoning about M-structure programs. We conclude in Section 6.

2 Background

2.1 The core functional language

The core of Id is a non-strict functional language with implicit parallelism[13]. It has the usual features of many modern functional languages, including a Hindley/Milner type inference system, algebraic types and definitions with clauses and pattern matching, and list comprehensions. An Id block, which has *letrec* semantics, consists of a set of bindings and a body expression following the keyword `In`, *e.g.*:

```
{ x = 1:y ;
  y = 2:x ;
In
  x }
```

Here, ":" is an infix list constructor, and the block evaluates to a cyclic list whose infinite unfolding is `1:2:1:2:`⋯ Note that non-strict evaluation allows this cycle to be created with mutually recursive bindings; such bindings are meaningless under strict evaluation.

In addition, Id has extensive facilities for arrays, including array *comprehensions* for the efficient construction of functional arrays. For example, the expression

```
{array (1,5) of
 | [j] = 0 || j <- 1 to 5 }
```

creates an array with index range 1 to 5, with every location containing 0.

One form of array comprehension useful for computing histograms is the *accumulator*. For example, suppose we wish to build a histogram of numbers ranging from 1 to 5 drawn from a list of numbers `xs`. The result is an array with index range 1 to 5, such that the j'th cell of the array contains the number of occurrences of j in `xs`. Here is the program:

```
{array (1,5) of
 | [j] = 0 || j <- 1 to 5
 accumulate (+)
 | [j] = 1 || j <- xs }
```

Before the `accumulate` keyword is an array comprehension that specifies the initial contents of the array (zeroes). The rest of the construct specifies that for each `j` in `xs`, we increment element j of the histogram by 1.

In array and accumulator comprehensions, no order is specified for the array-filling computations. However, note that array comprehensions can be non-strict, whereas accumulator comprehensions must be strict. In array comprehensions, each slot is assigned at most once, *i.e.,* there is a single transition from $\perp$ to non-$\perp$ (or "unevaluated" to "evaluated", in graph reduction terminology). Thus, the array can be returned as soon as it has been allocated, with consumers automatically blocking on $\perp$ if necessary. However, in accumulator comprehensions, each slot may go through several intermediate non-$\perp$ states, and we cannot be sure that a slot has reached its final value until all the accumulations have been performed. Thus, the array cannot be returned to a consumer until it has been fully computed[1].

2.2 Non-strict, Implicitly Parallel Semantics

We are interested in non-strict semantics, not only for its expressive power, but also because it admits more parallelism than strict semantics [14]. Many evaluation orders implement non-strict semantics, with lazy evaluation being the most popular. Id, on the other hand, implements non-strict semantics by *eager* evaluation of most expressions. A precise description of Id's operational semantics using rewrite rules is given in [1, 15]. The flavor of Id's semantics is given here to illustrate its expressive power and implicit parallelism. Briefly,

- In a block:

  ```
  { x1 = e1 ;
    ...
    xN = eN ;
  In
    eBody }
  ```

  all expressions (`e1`, ..., `eN` and `eBody`) are evaluated in parallel, and the value of `eBody` may be returned as soon as it is available (even if `e1`, ..., `eN` have not finished evaluating). Thus, the block has *letrec* semantics, with no implicit ordering on the evaluation of expressions, except as imposed by data dependencies.

- All expressions in an application `(e0 e1 ... eN)` may be evaluated in parallel. Suppose `e0` evaluates to a function `f` and we have a definition:

  ```
  def f x1 ... xN = eBody ;
  ```

  then the application can immediately be rewritten to:

[1] Array and accumulator comprehensions in Haskell were influenced by these array notations in Id[11].

```
{ x1 = e1 ;
  ...
  xN = eN ;
In
  eBody }
```

In other words, the function may be invoked as soon as `e0` is evaluated, and a result may be returned as soon as it is defined by the function, even if the arguments `e1`, ..., `eN` have not finished evaluating.

- In a conditional expression (`IF e1 THEN e2 ELSE e3`), `e1` is evaluated to a boolean value, after which *one* of the expressions `e2` or `e3` is evaluated.

Id's rewrite rules also take a precise position on *sharing* of computations and data structures, since this is also crucial in a language with state. Informally, an identifier such as `x1` above may not be substituted by `e1` until `e1` has been reduced to a variable or constant; this ensures that expressions do not get duplicated. Data structures are always referred to *via* labels, which may be regarded as abstract pointers. Thus, a constructor produces a single data structure that is shared among all its references.

Note that the operational semantics of Id are quite unique in that they are:

- Non-strict, but not lazy (blocks, constructors, and function applications may return values before any of their inputs are available), and
- Eager, but not call-by-value (evaluation of arguments is initiated whether they are needed or not).

As an example of eager, non-strict evaluation, consider the following. Given the function `f`

```
def f a b = (a+1,b+1) ;
```

we can evaluate the following block as sequence of rewrites:

```
{ (x,y) = f 3 x ;
In
   y}
```

First, the function application reduces to a nested block:

```
{(x,y) = { a = 3 ;
           b = x ;
         In
           (a+1,b+1) } ;
 In
   y}
```

In the rest of the sequence, nested block bindings are flattened into a single block, and tuple bindings are rewritten into identifier bindings. The block returns a value when the body identifier `y` is bound to a constant:

```
{ (x,y) = (a+1,b+1) ;  ==>  { x = a+1 ;  ==>  { x = 4 ;    ==>  5
  a     = 3 ;                 y = b+1 ;         y = b+1 ;
  b     = x ;                 a = 3 ;           a = 3 ;
In                            b = x ;           b = 4 ;
  y}                        In                In
                              y}                y}
```

This is perhaps an artificial example, but the cyclic binding establishes the point that *this is a non-strict language even though we do not use lazy evaluation.* To emphasize this further, we invite the reader to trace the evaluation of the function (due to Traub [18]):

```
def f b x y = { xx = if b then x  else yy ;
                yy = if b then xx else y  ;
              In
                xx + yy } ;
```

under the calls `(f True 10 20)` and `(f False 10 20)`. These expressions are undefined under strict semantics, but are perfectly meaningful in Id and other non-strict languages.

2.3 Non-strictness, yes, but why not lazy evaluation?

Consider the following function:

```
def f x = (e1, e2) ;
```

and suppose the expression `e1` contains a side-effect. In Id, we can immediately conclude that whenever `f` is called, the side-effect will occur.

In a lazy evaluator, on the other hand, we have to look at each application of `f` separately— the side-effect will occur only if the application is actually examined in a strict context and, further, that the first component of the resulting tuple is also examined in a strict context. For example, merely selecting the first component of the tuple and storing it into another data structure will not cause `e1` to be evaluated, so the side-effect will not occur. Thus, determining the conditions under which the side-effect will occur is as hard a problem as strictness analysis, requiring potentially global analysis. The presence of higher order functions only complicates matters further.

In Id, evaluation of an expression is controlled *only* by conditionals. Thus, determining the conditions under which a side-effect in a function `f` will occur requires only local analysis; we do not have to look at the contexts in which `f` is applied.

We believe that having such a *local* model of whether or not side-effects occur is essential in order to reason about programs with side-effects.

2.4 M-structures with *Take* and *Put* operations

As described, Id is a determinate language with no provision for side-effects. Its eager, non-strict semantics allows the efficient, parallel execution of functional programs. However, functional Id programs that share state suffer from the drawbacks alluded to earlier, *i.e.,* the loss of declarativeness and parallelism due to threading.

To support programs that share state, Id is extended with *M-structures*. M-structures are implicitly synchronized data structures that support atomic updates. We define two M-structure operations, called *take* and *put*, for reading and writing the components of an M-structure.[2]

Take and put operations are *implicitly synchronized*, *i.e.,* there is no separate notion of locks or semaphores. Rather, a state bit is associated with every M-structure element indicating whether it is *empty* or *full*. The empty state means that the element is currently taken and may not be read; the full state indicates that it is available. A *take* operation on a full element atomically reads its value and resets its state to empty, while a *take* on an empty element suspends. A *put* operation writes a new value to an empty (*i.e.,* taken) element;[3] if there are suspended *take* operations waiting for the element, one is resumed and the value communicated to it, and the element remains in the empty state. Otherwise, the value is written into the element and its state set to full.

For example, consider the following statement[4] that atomically increments an element of an M-structure array:

```
A![j] = A![j] + 1 ;
```

Here, the expression `A![j]` on the right hand side denotes a *take* operation on the j'th element of array `A`. This value is incremented by one; the expression `A![j]` on the left hand side denotes a *put* operation on the same element. Note that the strictness of the addition operator ensures that the *take* precedes the *put*. In general, the expression computing the value to be put into a cell should be strict in the value taken from the cell. Assuming this strictness, the M-structure synchronization guarantees that the cell update is atomic. For example, if k parallel computations attempt to execute the statement above, their access to the location will be properly serialized and the final value of the cell will be its original value plus k.

2.4.1 Explicit Sequencing

When writing parallel programs that share state, it is sometimes necessary to introduce explicit sequencing. For example, suppose we wish to reset an array location to 0 and return its old value, *e.g.,*

```
{ Aj = A![j] ;
  ---            % barrier
  A![j] = 0 ;
In
  Aj }
```

[2]An analogous pair of operations, called *E-fetch* and *E-store*, were developed independently by Milewski[12].

[3]A version in which multiple *put* operations are buffered has also been designed, allowing M-structures to act as producer-consumer channels. The examples in this paper will not use this feature.

[4]In Id, statements are included among a block's bindings

The first statement takes the old value out, and the second one puts the new value (0) back. The horizontal line is read as a sequentializing barrier, to ensure that the take occurs before the put. This is necessary since, by default, all components of a block are evaluated in parallel, and so there would be nothing to prevent the put occurring before the take. The barrier makes the put operation strict in the take.

A barrier in a block is more subtle than a simple sequencing form (the classical semicolon). It makes the statements following the barrier (including the body of the block) *hyperstrict* in the statements that precede the barrier. That is, no expressions following the barrier begin evaluation until all expressions before the barrier are completely evaluated, including procedure calls and all their expressions, recursively. This hyperstrict evaluation allows barriers to be used to sequence side-effects buried in procedure calls.

Unlike synchronization barriers in other parallel languages, an Id barrier is not a global barrier— it is entirely *local* to the block of bindings in which it appears, and it is reentrant, *i.e.,* each instance of the block has its own barrier. This not only makes them relatively easy to use, but they are also implemented very efficiently using "synchronization trees" (we do not have space to go into details here).

2.5 M-structures in Algebraic Types

As another example of an M-structure, consider a polymorphic list where each tail is updatable. The following type declaration defines such a list, where "`*0`" is a type variable and "`!`" denotes an updatable field (called an *m-field*):

```
type mlist *0 = MNil | MCons *0 !(mlist *0) ;
```

Although Id has pattern-matching on algebraic types, one usually does not use pattern-matching for types with m-fields, since pattern-matching and imperatives do not mix well. Thus, the fields of an algebraic type may also be selected using record syntax, with field names derived from the constructor and the field position. Ordinary fields, such as the head of the cons cell above, are selected using "dot" notation, *e.g.,* `x.MCons_1`. M-fields are taken using "bang" notation in an expression, as in `x!MCons_2`. A new value `v` may be put into an empty m-field using the statement:

```
x!MCons_2 = v
```

Type-checking ensures that m-fields and ordinary fields are accessed with the correct notation.

To compare the M-structure list with a functional list, consider the problem of inserting an integer `x` into a set `ys`, taking care not to insert duplicates. In the following functional solution, the set is represented as a list of integers and pattern matching is used:

```
typeof insert_f = (list int) -> int -> (list int) ;

def insert_f Nil    x = x:Nil
 |  insert_f (y:ys) x = if (x == y) then
                          ys
                        else
                          y:(insert_f ys x) ;
```

Using record syntax instead of pattern matching, `insert_f` is:

```
def insert_f ys x = if (Nil? ys) then
                      Cons x Nil
                    else if (x == ys.Cons_1) then
                      ys
                    else
                      Cons ys.Cons_1 (insert_f ys.Cons_2 x) ;
```

For comparison, the following solution uses M-structure lists:

```
typeof insert_m = (mlist int) -> int -> (mlist int) ;

def insert_m ys x = if (MNil? ys) then
                      MCons x MNil
                    else if (x == ys.MCons_1) then
                      ys
                    else
                      { ys!MCons_2 = insert_m ys!MCons_2 x ;
                      In
                         ys } ;
```

The first two conditional clauses of `insert_m` mimic the functional solution. In the third clause, we take the tail of the list, insert `x` into it and put it back, and we *return the original list*. Note that `insert_m` is strict in `ys` (due to the conditional), so the take and put are properly ordered. The advantage of this program over the functional version is that it takes $O(1)$ heap store per insertion, instead of $O(n)$, where n is the length of the list.

Under non-strict semantics, `insert_f` has the attractive property that in the last line, the new cons cell can be returned even while we are inserting `x` into its tail. Thus, multiple insertions can be "pipelined", *i.e.,* a second insertion can run closely behind the first, automatically synchronizing on empty slots.

Note that `insert_m` has the same kind of "pipeline" parallelism as `insert_f`. In the last line, non-strictness allows us to return `ys` immediately. If a second insertion is attempted immediately, it can run closely behind the first, blocking on empty fields (due to takes) in the same manner as `insert_f` blocks on empty fields.

However, there is an interesting subtlety here. Consider the following two near-identical fragments:

```
zs = insert_f ys x1 ;              zs = insert_m ys x1 ;
ws = insert_f zs x2 ;              ws = insert_m zs x2 ;
```

and assume that neither `x1` nor `x2` is present in `ys`. In the functional case, despite the pipelined behavior, `x2` is guaranteed to be inserted behind `x1`. The second insert_f can never "overtake" the first; if it tried to run faster, it would block at some point, waiting for a result from the first insert_f.

In insert_m, however, it is possible for the second insertion to overtake the first, inserting x1 behind x2. The reason is that in the parallel block in the last else clause, the first insert_m may return ys before it takes the MCons_2 field. Thus, when given ys, the second insert_m "races" with the first traversal to take of this field, and may potentially overtake it. If we want to prevent this overtaking, we have to delay returning ys until the take completes. This can be accomplished with a barrier:

```
def insert_m ys x = if ...
                      ...
                    else ...
                      ...
                    else
                      { ys' = ys!MCons_2 ;
                        ---
                        ys!MCons_2 = insert_m ys' x ;
                      In
                        ys } ;
```

Note that this expression does not wait for the put to take place before returning ys— that would make the insertion strict, eliminating pipelined behavior completely.

This subtlety highlights important differences between M-structures and their functional counterparts. First, atomicity is clearly a weaker property than determinacy. In the above example, both the M-structure and functional solutions are correct (neither make duplicate entries); however, the M-structure solution admits more parallelism through overtaking. Second, the M-structure solution can be modified to provide the sequencing property of the functional solution without the additional storage overhead. Finally, reasoning about the interaction of several concurrent operations is more difficult for M-structures than functional data structures. We address this issue in Section 5.

To conclude our discussion of the Id language and semantics, we make the following observation. In most approaches to parallel computing, one starts with a sequential language and adds parallel annotations. This is true even amongst functional languages, *e.g.,* the "future" annotation [8] and the "spark" annotation [7]. In contrast, we go in the opposite direction: we start with an implicitly parallel language and add sequentialization only where necessary. The degree to which we expose and exploit parallelism is thus much more aggressive.

2.6 Our Experiments

To determine the validity of our claims, we compare several Id programs written in a functional style (Approach A) against programs using M-structures (Approach C). These programs and our analysis are presented in the next two sections.

We evaluate the programs in several ways. First, we compare them subjectively in terms of declarativeness, *i.e.,* how well the structure of the program mirrors the structure of the problem. Second, we compare their efficiency in terms of storage use and number of instructions executed. Finally, their parallelism is compared. Since the parallelism exhibited by a program depends on many factors, this last measurement deserves some explanation.

Id programs compile into dataflow graphs, which constitute a parallel machine language (a partial order on instructions). The dataflow graphs respect the non-strict semantics and parallel evaluation order describe earlier in this section. The performance of these programs is measured on GITA, which is a dataflow simulator instrumented to take "idealized" performance measurements.[5] The GITA simulator executes dataflow graphs under the following idealized assumptions:

- No limit on the number of instructions that can be executed in parallel.
- An instruction is executed as early as possible, *i.e.,* as soon as all inputs are available.
- All instructions, including storage allocation "instructions", take 1 unit of time.
- No delay in communicating the output of one instruction to the input of another.

The purpose of idealized simulation is to give an upper bound on achievable performance, unconstrained by machine-specific characteristics, such as the number of processors, available memory, or scheduling policy. One important measure of idealized performance is the parallelism profile. This profile charts the number of parallel operations against time. Given the above assumptions, the parallelism profile exhibited under GITA describes the "inherent" parallelism of a program, *i.e.,* the program's limit of achievable parallelism. In this paper, we summarize these profiles as the average parallelism, which is the number of instructions divided by the time to completion.

Note that a major idealization in GITA is that each heap allocation takes one instruction. This hides the additional overhead of heap management encountered in a real system, such as garbage collection and initialization. In addition, storage management in a parallel system may also perform the task of load balancing. Thus, programs requiring more storage will, in fact, require more instructions than indicated by idealized performance measurements.

3 Histogramming a tree of samples

Consider the problem of histogramming numbers in the leaves of a tree. Given a tree T of numbers, where each number is in the range 1 to 5, the histogram H is an array of 5 elements such that H_i = the number of leaves containing number i.

[5] GITA is part of the "Id World" programming and simulation environment which has been in use at MIT and elsewhere since 1986.

This problem statement is a highly abstracted account of MCNP, an application program used by over 350 organizations in diverse industries such as oil exploration, automobiles, medical imaging, *etc.* [6] The program models the diffusion of subatomic particles from a radioactive probe through the surrounding material, using Monte Carlo simulation. For each original particle, the program follows it through events in its lifetime, such as motion in some direction, collision with a material boundary or another particle, *etc.* The tree structure arises because particles may split into two (and recursively, those particles may split again), after which the simulation follows each particle separately. When a particle finally dies, *i.e.,* reaches a leaf, some of its properties are collected in various histograms, such as final position and energy. MCNP has eluded easy parallelization in all conventional programming models, but is "embarassingly parallel", because the tree may have thousands of branches, and the events in a particle's life are decided purely locally, based on the toss of a coin.

In MCNP, no tree data structure is actually constructed. Rather, intermediate arguments and results are passed through the call/return tree. In our histogram example, the histogramming functions traverse a previously constructed tree, with the following type definition:

```
type tree = Leaf int | Node tree tree ;
```

The cost of creating this tree is not included in the measurements.

3.1 HA1: A Functional Solution

This program builds an initial array H0 containing 0 everywhere, and then performs a right-to-left traversal of the tree. At each leaf, the i'th element of the array is incremented by 1, where i is the number of the leaf. Since the functional solution may not use imperatives, incrementing this element requires building a new array, identical to the original, except that the i'th location is incremented. Here is the code:

```
typeof hist = tree -> (array int) ;

def hist T = { H0 = {array (1,5) of
                     | [j] = 0 || j <- 1 to 5 }
             In
               traverse T H0 } ;

typeof traverse = tree -> (array int) -> (array int) ;

def traverse (Leaf i)   H = incr H i
 |  traverse (Node L R) H = traverse L (traverse R H) ;

typeof incr = (array int) -> int -> (array int) ;
```

[6] Los Alamos National Laboratory is developing a comprehensive version of MCNP in Id under the direction of Olaf Lubeck.

```
def incr H i = {array (1,5) of
                | [j] = if (i==j) then H[j]+1 else H[j] || j <- 1 to 5 } ;
```

Critique

In principle, the increments can be done in any order. However, since each increment creates a new array, the array must be "threaded" sequentially through the traversal. Threading requires that the recursive calls to `traverse` be composed, which imposes a particular order on the increment operations. In the original MCNP problem, threading is even worse, since many arguments and results are passed through the call/return tree. The threading required by functional programming obscures the independent nature of the accumulations.

This program also suffers from storage overhead, since the histogram array is copied at each leaf of the tree. Each copy of the array requires 5 loads and stores, while the problem requires only a single load and store per increment. Note that techniques for eliminating copying in functional programs, such as single-threadedness analysis and abstract reference counting, might determine that all the updates could occur in place. However, these analyses assume a *sequential* order of execution, and so preclude parallel execution.

3.2 HA2: A Functional Solution using Accumulators

In this example, we build an accumulator array with each element initialized to zero. The accumulation draws from a list produced by `traverse`, which collects all the numbers in the leaves of the tree. Here is the code:

```
typeof hist = tree -> (array int) ;

def hist T = {array (1,5) of
              | [j] = 0 || j <- 1 to 5
              accumulate (+)
              | i = 1 || i <- traverse T } ;

typeof traverse = tree -> (list int) ;

def traverse T = aux T Nil ;

typeof aux = tree -> (list int) -> (list int) ;

def aux (Leaf i)   is = i:is
 |  aux (Node L R) is = aux L (aux R is) ;
```

Critique

The use of an accumulator for the histogram eliminates the need to copy the array. In fact, the accumulator is quite declarative and modular: for each i drawn from the list of leaf numbers, the i'th element is incremented by one. The functions `traverse` and `aux` still require threading to construct the list of leaf numbers. However, non-strictness allows this list to be constructed in parallel and "piped" into the accumulator.

Although an improvement, the intermediate list read by the accumulator incurs storage overhead and overspecifies the order of accumulations. Accumulators use lists for modularity and composition; eliminating the list through compiler analysis requires global analysis to thread the accumulator through the generator function and its descendants.

The use of accumulators in an early version of the MCNP program corroborates these observations: the intermediate list complicated the program and introduced inefficiencies. The M-structure solution, presented next, retains the declarativeness of accumulators and eliminates the need for an intermediate list.

3.3 HC: An M-structure Solution

In this solution, the histogram H is represented as an M-array initialized to zero. The `traverse` function recursively passes H down to all leaves in parallel. Each leaf (containing i) atomically increments element H_i by taking the element, adding one, and putting it back. A sequential barrier is used to ensure that H is not returned until `traverse` completes. (Recall that this barrier is implicit in accumulators, since they may not return until all accumulations have been performed.) The M-structure program is below:

```
typeof hist = tree -> (m_array int) ;

def hist T = { H = {M_array (1,5) of
                     | [j] = 0 || j <- 1 to 5 } ;
                _ = traverse T H ;
               ---                                   % sequential barrier
              In
                H } ;

typeof traverse = tree -> (m_array int) -> void ;

def traverse (Leaf i)   H = { H![i] = H![i] + 1 }
  | traverse (Node L R) H = { _ = traverse L H ;
                              _ = traverse R H } ;
```

Bindings of the form "`_ = e`" are used to execute `e` for its side-effects and discard its result. Let blocks with no body expression return a meaningless value of type `void`, and are executed solely for side-effect.

Critique

This program requires no threading since the accumulation is performed by side-effects. Thus, the recursive calls to `traverse` are not composed, and the order of accumulations is not overspecified. The atomicity of each accumulation is is *locally* specified by the take and put operations and their implicit synchronization. In contrast to the functional programs, only storage for the result histogram is required, and each accumulation requires only a single load and store (take and put).

3.4 Other Functional Solutions

Several techniques for improving the performance of parallel and functional programs have been developed, such as parallel combining trees and tree-structured arrays. These techniques were used in three other functional solutions, which are summarized below:

HA3: Tree Accumulation. The program recursively descends in parallel to every leaf. At a leaf containing, say, 2, we produce the array `[0 1 0 0 0]`. At each node, we add the arrays from the two subtrees and return the new array. Thus, the top node produces the sum for the whole tree. Although this eliminates threading, its storage requirements are double that of HA1.

HA4: Parallel Traversal. The entire tree is traversed five times in parallel (once for each element in the histogram). The i'th traversal uses parallel tree accumulation to sum the number of leaves containing i. Since this accumulation computes an integer, it requires no additional storage. Finally, the histogram is constructed by storing the five sums into an array, which is returned. Although this has no threading and minimal storage, it executes a large number of instructions, because the entire tree is traversed for each element of the histogram.

HA5: Tree-Structured Histogram. The program is a modification of HA1, replacing the histogram array by a balanced tree. This reduces memory and instruction overhead by copying a single path of the tree ($O(\log b)$ instead of $O(b)$, where b is the number of elements in the histogram). Note that copying a path requires several memory allocations rather than one, as well as conditionals to route each increment to the appropriate leaf. Therefore, this solution is less efficient than HA1 for small histograms. Even with large histograms, threading still obscures the code and overspecifies the order of accumulations.

3.5 Experimental results

We ran the above programs on a full binary tree of depth 10, *i.e.,* 1024 leaves with an (almost) equal number of 1's, 2's, ... and 5's. The results are shown below:

Program version	Total instructions executed	Critical path	Avg. parallelism	Heap store used (words)
HA1	199,757	1,452	138	9,250
HA2	72,847	7,324	10	2,100
HA3	325,472	238	1368	18,423
HA4	266,223	195	1365	9
HA5	253,370	28,833	9	11,301
HC	60,496	589	103	9

3.5.1 Observations

Declarativeness: We believe the M-structure solution HC is more declarative than the functional solutions, since it does not overspecify accumulation order by threading.

Instruction count: HC requires the fewest number of instructions. The instruction overhead of copying in solutions HA1, HA3, and HA5 is significant when compared to HA2 (accumulators) and HC. The high instruction count in HA4 is due to the duplicate traversals of the tree.

Heap Store Used: Copying intermediate data structures in the functional solutions dominate their storage profiles. Imperative updates, as used in HC, allow storage use to be reduced by three orders of magnitude.

Parallelism: HA3 and HA4 have more parallelism than the other programs, but this is mostly due to redundant computation, such as copying arrays or multiple tree traversals. The poor parallelism in HA2 and HA5 comes from overserialization: the intermediate list in the case of accumulators, and copying the path of the tree in HA5. The parallelism in HC is limited by synchronization of take and put instructions. Because they are atomic, the number of parallel increment operations is bounded by the size of the histogram. Parallelism in this example exceeds 5 because these increments are overlapped with the tree traversal.

In summary, the M-structure solution to the histogramming problem is highly declarative, and combines instruction efficiency and low storage use with significant parallelism.

4 A Graph Traversal Problem

For a second comparison of functional and M-structure programs, consider a simple graph traversal problem. Suppose we are given a directed graph structure, *i.e.,* an interconnection of nodes:

```
type gnode = GNode int int (list gnode) ;
```

The first `int` field contains a unique identifier for the node, the second `int` field contains some numeric information, and the list of nodes represents the list of neighbors (possibly empty) of the current node.

The problem is this: given a node A in a graph, compute `rsum(`A`)`, the sum of the integer information in all nodes reachable from A.

The unique identifiers in the nodes highlight an important difference between graphs and trees: graphs can have can have shared substructures and cycles. Graph traversals need to take this into account to avoid repeated traversals of nodes and subgraphs. Therefore, traversal programs must be able to test equality between nodes. The notion of object equality differs between functional and imperative languages, but is not central to the issues in this paper. For clarity, the unique ID field in the graph nodes will be used to determine node equality.

Traditional graph traversal algorithms rely on "marking" visited nodes to avoid repeated traversals. For familiarity, we will begin this section with two M-structure programs that use imperative operations to mark visited nodes. This is followed by the functional solution and experimental results.

4.1 GC1: A Simple M-structure Solution

The traditional imperative solution to this problem involves extending the `node` type to contain a *mark* field, which is used to avoid repeated traversals of shared subgraphs. We assume that the mark field is initially set to `False`. The following program expresses this solution:

```
type gnode = GNode int int (list gnode) !bool ;

def rsum nd = if (marked? nd) then
                0
              else
                nd.GNode_2 + sum (map rsum nd.GNode_3) ;

def marked? nd = { m = nd!GNode_4 ;        % take the mark field
                  ---                      % sequentialize
                   nd!GNode_4 = True ;     % put True in mark field
                 In
                   m } ;
```

Note that in this solution, the unique ID field is not required.

Critique

The solution is very clear. The only subtle point is the atomicity of the `marked?` predicate. The value of the mark field is taken and returned, and set to `True` regardless of its previous value. Since the mark field is set to `True` the first time the node is visited, the node will be counted only once. As described earlier, the serialization barrier between the take and put guarantee that they happen in the correct order.

While this solution corresponds to most textbook algorithms, it has two drawbacks:

Mark initialization: We assumed that the marks in the graph were initialized to `False` before the traversal began. But, how do we achieve this? Traditionally, we have some *independent* access to all the nodes, such as an array or set of all nodes, and we iterate through this collection, resetting all marks. Note that resetting the marks cannot be overlapped with the traversal algorithm—this easily doubles the cost of the algorithm. Further, if the graph is large compared to the region traversed, resetting all marks may be more expensive than the traversal.

Multiple traversals: Suppose we are given two nodes A and B and want to compute the reachable sum from each of them, in parallel. GC1 cannot be used for this, since the marks from the A and B traversals may interfere, *i.e.,* nodes marked by A's traversal will not be counted by the traversal from B, and vice versa.

These drawbacks can be overcome by a simple extension of the "mark field" idea. Here, each traversal carries an additional parameter, a unique *traversal identifier*. Rather than marking nodes with a boolean, each node contains a list of traversal identifiers indicating who has visited the node. The `marked?` predicate checks and updates this list.

This solution allows multiple traversals to occur in parallel, since they have distinct traversal IDs and will not interfere. Also, node marks do not have to be reset, since a brand new traversal ID is issued for each traversal. However, the performance of this solution worsens as more and more traversals are performed, creating unused IDs that slow the `marked?` predicate and occupy storage. Because of this weakness, we will not pursue this idea any further.

4.2 GC2: A Solution Allowing Multiple Traversals

A better solution that allows multiple traversals uses a separate `visited` table to keep track of nodes already traversed. Rather than marking nodes directly, this table contains the unique IDs of visited nodes, and is updated as each new node is encountered. The original definition of graph nodes (without a mark field) is used:

```
type gnode = GNode int int (list gnode) ;
```

Here is the code for the reachable sum:

```
def rsum nd = { visited = mk_empty_table () ;

                def aux (GNode x i nbs) =
                  if (member?_and_insert  visited  x) then
                    0
                  else
                    i + sum (map aux nbs) ;
              In
                aux  nd } ;
```

The `member?_and_insert` function is analogous to the `marked?` function in GC1: it returns a boolean indicating whether `x` is present in the `visited` table, inserting `x` in the table if it is not.

Critique

The version of rsum given in GC2 is quite similar to GC1. There is no threading, and marking code is localized. The main difference between the programs is the modularity supplied by the visited table. It is shared by all calls to aux, but does not require threading or modification to the graphs. Further, its storage can be released after the traversal.

4.2.1 A Parallel Hash Table

We can implement the visited table in a number of ways. A parallel hash table is a good candidate, since it allows $O(1)$ access to the elements of a dynamically created set. An empty hash table of size N (for some suitable constant N) is constructed using an m-array comprehension:

```
def mk_empty_table () = {M_array (1,N) of
                        | [j] = MNil || j <- 1 to N} ;
```

Each bucket of the array is initialized to an empty mlist.

As with the marked? function in the GC1 solution, the member?_and_insert function must execute atomically to avoid duplicate traversals. This function tests whether an integer x is in the hash table and, if not, inserts it:

```
typeof member?_and_insert = (m_array (mlist int)) -> int -> bool ;

def member?_and_insert  table  x =
  { (l,u) = bounds table ;
    j = hash l x u;
    (b,ys') = member?_and_insert_list table![j] x ;
    table![j] = ys' ;
  In
    b } ;
```

In the first line in the block we extract l and u, the lower and upper index bounds of the table, and in the second line we hash the given integer index x into an index j in the range l to u (using some unspecified hash function). Then, we take the set of integers at index j and uses the member?_and_insert_list function to test if x is in that set and insert it if not. Finally, the (possibly) new set is put back and the boolean result returned.

Note that the list function that tests for membership, followed by conditional insertion, is *not* a simple composition of the member and insert functions on m-lists. The elements of the list must be tested and inserted in a single scan of the list to ensure atomicity. Otherwise, a node may be seen as unmarked by two processes, resulting in redundant traversals.

We modify our insert_m function from Section 2.5 to perform this simultaneous test for membership and insertion:

```
typeof member?_and_insert_list = (mlist int) -> int -> (bool,(mlist int)) ;

def member?_and_insert_list ys x =
  if (MNil? ys) then
    (False, MCons x MNil)
  else if (x == ys.MCons_1) then
    (True,ys)
  else
    { (b,ys') = member?_and_insert_list ys!MCons_2 x ;
      ys!MCons_2 = ys' ;
    In
      (b,ys) } ;
```

Note that the hash function can be applied to two integers `x1` and `x2` in parallel. If they hash to different indices, there is absolutely no interference between the two calls. Even if they hash to the same index, they may "pipeline" as discussed earlier.

GC2': A Mark Array

We can improve the implementation of the `visited` table even further if we know that the unique identifiers in the graph nodes are in some contiguous range, say 1 to N. In this case, the table can be implemented as an array of booleans, instead of a hash table, *i.e.,*

```
def mk_empty_table () = {m_array (1,N) of
                         | [j] = False || j <- 1 to N} ;

def member?_and_visited table x = { b = table![x] ;
                                    ---
                                    table![x] = True
                                  In
                                     b } ;
```

Note that in the pure functional program described next, the $O(1)$ access time of the hash and mark tables cannot be exploited. Implementing the `visited` table as a balanced binary tree is the best we can do.

4.3 GA1: A Functional Solution

Since nodes cannot be imperatively marked, the functional solution also uses a `visited` table to keep track of nodes that have already been traversed. This table is threaded through the traversal, as shown in the following program:

```
def rsum nd = { visited = TEmpty ;

                def aux (s,visited) (GNode x i nbs) =
                    if (member? x visited) then
                      (s,visited)
                    else
                      {  visited' = insert visited x ;
                      In
                         foldl  aux  (s+i,visited')  nbs } ;

                (s,visited') = aux (0,visited) nd
              In
                s } ;
```

The function foldl is similar to map, but passes partial results (the partial sum and visited table) to each successive call of aux. Here is the code for foldl:

```
def foldl f z Nil    = z
 |  foldl f z (x:xs) = foldl f (f z x) xs ;
```

The visited table can be implemented as an ordered binary tree, with $\log n$ time for the member? and insert functions, assuming the tree is balanced:

```
type tree = TEmpty | TNode int tree tree ;

def member? TEmpty        x = False
 |  member? (TNode y L R) x = if (x == y) then
                               True
                              else if (x < y) then
                                member? L x
                              else
                                member? R x ;

def insert TEmpty        x = TNode x TEmpty TEmpty
 |  insert (TNode y L R) x = if (x == y) then
                               (TNode y L R)
                             else if (x < y) then
                               (TNode y (insert L x) R)
                             else
                               (TNode y L (insert R x)) ;
```

Critique

The rsum program is obscured by the "threading" the visited table through the traversal. Again, threading overspecifies the order in which the traversal is made; here, the order of insertions is determined by the function foldl. Conceptually, one imagines all outward edges from a node being explored in parallel, with a shared subnode being explored only by the first traversal that happens to arrive there (we don't care which one).[7]

[7] Readers who are familiar with parallel graph reduction will recognize that this is exactly what happens in a parallel graph reducer— a shared node is evaluated by the first process that arrives there, which also marks it as "in progress",

Note that here, unlike the histogram example, sharing is critical to ensure a polynomial time algorithm. Therefore, threading is unavoidable in purely functional solutions. In more complex problems, the threading required by functional programming further complicates the solution, adding extra parameters and return values and imposing unnecessary serialization.

The functional solution also introduces storage overhead. Each `insert` rebuilds the `visited` table along the path from the newly inserted leaf back to the root, allocating new tree nodes along the way. The total storage cost for the table over the complete graph traversal can therefore vary from $O(n \log n)$ to $O(n^2)$, depending on how well the tree is balanced. Including tree rebalancing operations introduces overhead that may be recouped over many operations.

4.4 Experimental Results

We ran tests on our simulator, running four versions of `rsum` on four graphs. The graphs each contained about 512 nodes, but differed widely in their topologies (amount of sharing). The four versions of `rsum` were:

GA1	The functional program with a `visited` table implemented as an ordered binary tree.
GC2	The M-structure program with a `visited` table implemented as a hash table of size 523.
GC2'	The M-structure program with a `visited` table, implemented as an array of booleans.
GC1	The M-structure program using mark fields in graph nodes.

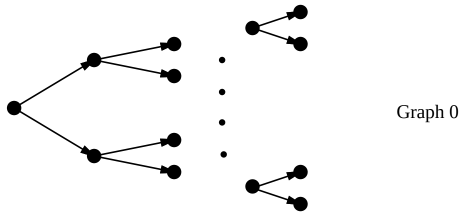
For the functional version, the unique identifiers in the graph nodes were randomly generated so that the binary tree `visited` table would be roughly balanced, in order to show the numbers for the functional program in the best possible light. Random ID generation was also used for the hash and marks version, though it does not really matter for these versions. For the "array" version, IDs were integers in the range of 1 to n, the number of nodes in the graph, and the array of marks had the same index range.

The fundamental difference in heap store use for the four programs is in the `visited` tables. We modified the programs slightly from the text of the paper in order to measure this, *i.e.,* to separate heap usage for the `visited` table from heap usage for closures in higher order functions and tuples for multiple results. These changes do not affect the total instruction counts or parallelism very much.

The cost of originally building the graphs (storage and instructions) is not included because the graphs were built in a separate invocation before the application of `rsum`.

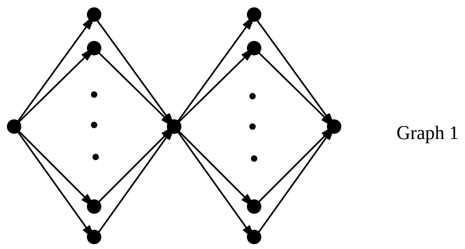
Figures 1 through 4 show the four graphs and the measurements for the four programs.

so that later-arriving processes will not duplicate the work.



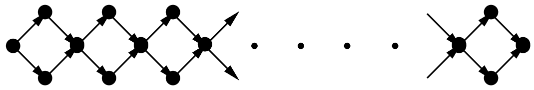
Program version	Total instructions executed	Critical path	Avg. parallelism	Heap store used (words)
GA1	480,123	90,605	5.3	15,934
GC2	102,244	4,280	23.9	2,072
GC2'	55,843	2,743	20.4	524
GC1	48,002	412	116.5	0

Figure 1: Reachable Sum on Graph0 (511 nodes).



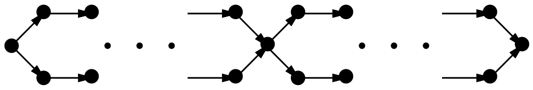
Program version	Total instructions executed	Critical path	Avg. parallelism	Heap store used (words)
GA1	594,323	124,276	4.8	17,305
GC2	177,062	9,271	19.1	2,959
GC2'	95,501	9,102	10.5	524
GC1	84,078	3,728	22.6	0

Figure 2: Reachable Sum on Graph1 (515 nodes).



Program version	Total instructions executed	Critical path	Avg. parallelism	Heap store used (words)
GA1	587,873	123,434	4.8	16,998
GC2	130,188	22,527	5.8	2,068
GC2'	68,933	12,799	5.4	524
GC1	59,902	11,596	5.2	0

Figure 3: Reachable Sum on Graph2 (511 nodes).



Program version	Total instructions executed	Critical path	Avg. parallelism	Heap store used (words)
GA1	549,169	103,917	5.3	18,550
GC2	103,456	28,844	3.6	2,076
GC2'	56,385	10,359	5.4	524
GC1	48,518	8,808	5.5	0

Figure 4: Reachable Sum on Graph3 (515 nodes).

4.4.1 Observations

Parallelism: Graph0 and Graph1 have a high degree of parallelism, *i.e.,* there is much branching without sharing; Graph2 is not very parallel (much branching, but also much sharing), and Graph3 is also not very parallel (not much branching). Nevertheless, the functional program is unable to exploit the parallelism in Graph0 and Graph1, because the threading of the `visited` table forces it to be practically sequential. All the other programs, however, were able to exploit the additional parallelism of those graphs.

Instruction count: The instruction counts for the functional program GA1 is highest, by over a factor of 10 from the most efficient version (marks).

Heap Store Used: The $O(n\log(n))$ heap store cost for the `visited` table in the functional program is apparent— it uses close to 10 times more heap store than its nearest rival (the hash program GC2). GC2 takes about 523 words for the hash array, plus 3 words (one cons cell) for each of the about 512 entries in the table, making a total of a little over 2000 words. The mark array GC2' program takes exactly 524 words for the array of booleans. GC1, which marks the nodes directly, allocates no store, but we should remember that we have already paid one word extra per graph node for the mark field, *i.e.,* a total of about 512 words.

Again, we point out that the functional program is being presented in it best possible light because (a) we are charging only one instruction for each heap allocation, and (b) we have set up the IDs to keep the `visited` tree balanced.

5 Discussion

In this section, our work on M-structures is put in the context of other work. First, we describe our preliminary experiences validating our simulation results on a real multiprocessor. Next, M-structures are briefly compared to other non-deterministic extensions to functional languages. Finally, a method for reasoning about M-structure programs is outlined.

5.1 Validation

In order to ensure that our simulations are reasonably accurate, we have also run all the above programs on Monsoon, a parallel machine with 64-bit, 10MIPS dataflow processors and 4 MWord I-structure memory units[16]. These processors are being built in partnership with Motorola, Inc. Our programs ran on a two-node prototype; a sixteen-node machine is expected to be available in June, 1991. The major unaccounted cost in our simulation is that of heap allocation, since the simulator counts each heap allocation as a single instruction. In Monsoon, therefore, instruction counts are higher than in our simulations, but the overall trends are the same (the Monsoon figures are too preliminary to publish at this time).

5.2 Nondeterminism in Other Languages

Since Approach C admits non-deterministic, parallel programs, it is interesting to compare it to Approaches A and B extended to allow non-determinacy. Several non-deterministic extensions have been proposed for Approach A, such as amb[19], streams[17], and managers[2, 4]. To retain the flavor of functional programming, these solutions are *process-oriented*: many processes share information over an *implicit* communication channel. This channel merges updates and synchronizes accesses. Each process can then be viewed as a state transformation function that maps successive values on the shared channel. Thus, each process can be analyzed as a function, with only the aggregate behavior producing non-determinism.

Approach B allows imperative operations but ensures determinacy through a sequential, strict control paradigm. Extensions for parallelism, such as fork and join in Scheme and ML, require explicit synchronization constructs to be used by the programmer. Proving such programs correct is difficult, and much of the original declarativeness of the program is destroyed.

M-structures are a *data-oriented* approach to non-determinism, which provides the efficiency of Approach B. In addition, M-structures provide the implicit synchronization, which guarantees the atomicity of operations on individual elements of data structures. This fine-grained atomicity allows for substantial parallelism in programs that share data without additional synchronization complexity. The result is highly declarative, efficient programs.

5.3 Reasoning about M-structure Programs

The attractiveness of an implicitly parallel, non-strict language like Id is that programmers can write parallel, determinate programs without annotations or explicit synchronization. Reasoning about determinate Id programs follows in the traditions of functional programming: functions map expressions to values, and correctness proved by equational reasoning.

M-structures introduce indeterminacy, and thus cannot use equational reasoning. Instead, the correctness of an M-structure program is related to the notion of serializability in databases. That is, correctness is defined in terms of the history of values held in an M-structure cell. In a correct program, the sequence of cell values is a member of an allowed set of sequences. Serializability is one notion of correctness, requiring that any parallel execution corresponds to some interleaving of its update operations. Serializability ensures that M-structure updates appear atomic and sequential.

Determining the possible histories of an M-structure cell requires an operational understanding of Id. The set of M-structure histories is determined by the possible orderings of take and put operations on the cell. Since Id is an implicitly parallel language, the order of operations is a partial order, formed by a precedence relation $\prec$ between operations. That is, given operations a and b in program P, if $a \prec b$, then a will execute before b under any correct execution of P. Precedence between two operations is determined by three things: data dependence, control constructs, and data structures.

Data dependence defines precedence between strict operators. For example,

```
{i = a+b;
 In
   i*i};
```

yields `+`$\prec$`*`, since the multiplication is strict in the result of the addition.

Control constructs define precedence between a predicate and the selected expression. For example, in the following `if` expression,

```
 if i==0 then
   i+1
 else
   i+2
```

the equality test precedes the addition in both arms, *i.e.,* `==`$\prec$`+`.

Finally, M-structures define a precedence relation, since a take is always preceded by some put. A formal description of precedence has been developed by the authors, based on operational semantics for Id given in [1].

We now outline a correctness proof of the histogram program HC that uses M-structures. We wish to show that each cell in the histogram contains the sum of the number of leaves in the tree with its index. We can prove this by showing that the histogram is initialized to 0, that each leaf is reached once, and that each update is atomic. Since the tree traversal is a simple recursion, the second step can be proved by traditional inductive means.

First, let us address the atomicity of the updates. When a leaf is reached, the expression `H![i] = H![i]+1` is evaluated. This corresponds to a take, an addition, and a put. Put and addition are strict in their arguments, therefore `take`$\prec$`+`$\prec$`put`. M-structure synchronization guarantees that exactly one take operation will get a value after a put. This value will be incremented and put back, as defined above, potentially satisfying another take. This continues until all increments complete. Thus, the precedence of this expression, and M-structure synchronization yield some interleaving of atomic updates.

To show that M-structure cells are initialized correctly. When an M-structure is allocated, its elements are initially empty; the constructor (the array comprehension) initializes these elements by putting a value in each element. In this case, the allocate is followed by a put with the value 0. Since any take must be preceded by a put, no increment operation will begin until the cell has been initialized.

Therefore, the history of histogram cell i is

`alloc`$\prec$`put 0`$\prec$`take`$\prec$`+`$\prec$`put`$\prec$`take`$\prec$ `+`$\prec$`put`$\prec \cdots \prec$`put`

which always corresponds to a serialization of the update operations. Since addition is commutative and associative, serialization is sufficient to determine that the cell has the correct count after the last put, regardless of the order of the increments. As expected, eager evaluation allows this proof strategy to use local assertions about initialization and update operations.

Note that M-structure synchronization enforces this precedence relation even if a take operation is invoked before a put operation, or several takes are issued simultaneously. The implicit synchronization provided by M-structures allows the set of all possible interleavings to be reduced to only the set allowed by the precedence relation. Proving the correctness of this set depends on the application.

Reasoning about M-structure programs involving more than a single cell is more complex, but follows the same principles. In this case, atomicity involves synchronizing several M-structure operations to prevent processes from interfering and to avoid deadlock. To aid the development of such programs, the authors are developing constructs for encapsulating M-structure operations to ensure atomicity. Such encapsulation has been shown to be useful elsewhere [6, 9] for reasoning about parallel programs that share state.

6 Conclusion

Although the results presented in this paper may be viewed as a recommendation to use Approach C in programming, it may also be viewed as a challenge for the functional programming community to come up with new purely functional notations and optimization analyses that match the declarativeness and efficiency of Approach C. There is some precedent for this kind of development. Some years ago, we proposed a non-functional construct called "I-structures", showing how they cleanly overcame certain limitations in expressive power and efficiency in functional languages [5]. This stimulated much debate and research, leading to the purely functional "array comprehension" notation in Id, which eliminates much of the need for I-structures. Perhaps this paper on M-structures can serve a similar role and lead to new developments in pure functional languages.

Acknowledgements: This paper describes research performed at the Laboratory for Computer Science of the Massachusetts Institute of Technology. Funding for the Laboratory is provided in part by the Advanced Research Projects Agency of the Department of Defense under the Office of Naval Research contract N00014-89-J-1988. Paul Barth is supported by a fellowship from Schlumberger Technology Corporation.

References

[1] Z. M. Ariola and Arvind. P-TAC: A Parallel Intermediate Language. In *Proceedings of the Fourth Conference on Functional Programming Languages and Computer Architecture, London*, pages 230–242, September 1989.

[2] Arvind and J. D. Brock. Resource Managers in Functional Programming. *Journal of Parallel and Distributed Computing*, 1(1), June 1984.

[3] Arvind, D. Culler, and G. Maa. Assessing the Benefits of Fine-Grained Parallelism in Dataflow Programs. *International Journal of Supercomputing Applications*, 2(3), 1988.

[4] Arvind, K. P. Gostelow, and W. Plouffe. Indeterminacy, Monitors and Dataflow. *Operating Systems Review (Proceedings of the Sixth ACM Symposium on Operating Systems Principles)*, 11(5), November 1977.

[5] Arvind, R. S. Nikhil, and K. K. Pingali. I-Structures: Data Structures for Parallel Computing. *ACM Transactions on Programming Languages and Systems*, 11(4):598–632, October 1989.

[6] F. W. Burton. Encapsulating Non-determinacy in an Abstract Type with Determinate Semantics. *Journal of Functional Programming*, 1(1), January 1991.

[7] C. Clack and S. L. Peyton Jones. The Four-Stroke Reduction Engine. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming, Cambridge, Mass.*, pages 220–232, August 4-6 1986.

[8] R. H. Halstead. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–539, October 1985.

[9] C. A. R. Hoare. Monitors: An Operating System Structuring Concept. *Communications of the ACM*, 10(10):549–557, October 1974.

[10] P. Hudak. A Semantic Model of Reference Counting and Its Abstraction. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, Computers and Their Applications, chapter 3, pages 45–62. Ellis Horwood Limited, Chichester, West Sussex, England, 1987.

[11] P. Hudak and P. Wadler (editors). Report on the Programming Language Haskell, A Non-strict Purely Functional Language (Version 1.0). Technical Report YALEU/DCS/RR777, Yale University, Department of Computer Science, Apr. 1990.

[12] J. Milewski. Functional Data Structures as Updatable Objects. *IEEE Transactions on Software Engineering*, 16(12):1427–1432, December 1990.

[13] R. S. Nikhil. Id (Version 90.0) Reference Manual. Technical Report CSG Memo 284-1, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, USA, July 1990.

[14] R. S. Nikhil. The Parallel Programming Language Id and its Compilation for Parallel Machines. In *Proc. Workshop on Massive Parallelism, Amalfi, Italy, October 1989*. Academic Press, 1990 (to appear).

[15] R. S. Nikhil and Arvind. *Programming in Id: a parallel programming language*. 1990. Textbook on implicit parallel programming. In preparation.

[16] G. M. Papadopoulos and D. E. Culler. Monsoon: An Explicit Token Store Architecture. In *Proc. 17th. Intl. Symp. on Computer Architecture, Seattle, WA*, May 1990.

[17] W. Stoye. Message-Based Functional Operating Systems. *Science of Computer Programming*, 6:291–311, 1986.

[18] 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.

[19] D. Turner. Functional Programming and Communicating Processes. In *Proceedings of PARLE: Parallel Architectures and Languages, Europe, Volume II, Eindhoven, The Netherlands, Springer-Verlag Lecture Notes In Computer Science, Volume 259*, pages 54–74, June 1987.

Contents