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

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[1]

Paul Barth, Rishiyur S. Nikhil and Arvind

MIT Laboratory for Computer Science

March 18, 1991

Abstract

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. In addition, functional programs have *implicit* parallelism: simple compiler analysis can identify expressions that may evaluate in parallel without additional programmer annotations. However, when functional programs use shared data structures, both their declarativeness and parallelism can be compromised. This is because the shared data must be "threaded" through the function call tree, in order to ensure determinacy. Further, updates to the structure may require copying it. One way to reduce threading and copying is to introduce imperative operations, as in ML and Scheme. To provide a meaningful semantics in the presence of side-effects, these languages require strict, sequential evaluation—thus eliminating the benefit of parallelism.

In this paper we present M-structures, which are imperative data structures with implicit synchronization. M-structures have been added to Id, an implicitly parallel, declarative language with non-strict semantics. We argue that Id programs that use M-structures to share state improve on their functional counterparts in three ways: M-structure programs are (1) more *declarative*; (2) more *parallel*; and (3) more *storage efficient*. Points (1) and (2) are surprising, in that they contradict the conventional wisdom about functional programs stated above. We demonstrate our claims by studying two algorithms that share state: computing the histogram of a tree of samples, and traversing a graph. These programs are written both functionally and with M-structures, and evaluated for declarativeness, parallelism, and efficiency. Performance analysis is given from both idealized simulation and execution on Monsoon, a 10MIPS dataflow processor.

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, functional programming is widely viewed as having some major weaknesses. One such weakness is that certain programs cannot be expressed naturally. 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 is that it is difficult to implement functional languages 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.

[1] 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.

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 [13]).

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 updateable data structures manipulated with implicitly synchronized imperatives called *take* and *put* operations, 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) brings up 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], which is beyond the scope of this paper.

In this paper, we will demonstrate our claims by comparing Approach A (functional programs) and Approach C for two example applications. These are analyzed in terms of declarativeness and performance. Performance is evaluated experimentally by running the programs on GITA, an idealized parallel simulator and Monsoon, a 10MIPS dataflow computer. 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 *take* and *put* extensions, and our parallel simulator. Sections 3 and 4 explore the two example programs, respectively, in some detail. We conclude in Section 5 with some perspective.

> *Please Note*: Monsoon arrived in the lab shortly before the time of this writing. Thus, performance measurements have been taken on only the first example, and few optimizations have been made. During the time of review, a complete set of measurements (on larger data sets) will be taken. These will be reported on in the published paper.

2 Background

2.1 The core functional language

The core of Id is a non-strict functional language with implicit parallelism[12]. It has the usual features of many modern functional languages, including a Hindley/Milner type system, algebraic types and definitions with clauses and pattern matching, and list comprehensions. LETREC blocks in Id are written as shown in this example:

```
{ 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: ....`

In addition, Id has extensive facilities for arrays. The "array comprehension" is a constructor for an array. For example,

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

is an expression that constructs an array with index range 1 to 5, with every location containing 0.

An extension to array comprehensions is the "accumulator comprehension". For example, given a list of numbers `xs` in the range 1 to 5, suppose we wish to build a histogram of these numbers, *i.e.*, 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 accumulate

```
A[j] := A[j] + 1 % pseudocode; this is not Id syntax
```

where each operation is done atomically.

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[2].

2.2 Implicitly Parallel Evaluation Order

We are interested in functional languages with non-strict semantics, not only for their expressive power, but also because non-strict semantics admits more parallelism than strict semantics[3]. To implement non-strict semantics, there are many choices for evaluation order, and lazy evaluation is the most popular one. However, lazy evaluation does not combine well with imperative extensions. *In lazy evaluation, it is difficult to reason about whether an expression will get evaluated or not.* This makes it very difficult to reason about whether an imperative construct somewhere in the program will get executed or not which, in turn, makes it difficult to reason about programs in this style.

In Id, therefore, we choose a different evaluation order, while retaining non-strict semantics. 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 though `e1`, ..., `eN` may not have finished yet). 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. 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 though `e1`, ..., `eN` may not have evaluated yet.[4]

- 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.

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

- Non-strict, but not lazy (blocks and function and constructor applications may return values before any of their inputs are available), and

[2] Historical note: array and accumulator comprehensions in Haskell were inspired by these array notations in Id[11].
[3] See [13] for a demonstration of this point.
[4] This parallel evaluation rule holds when `e0` is a constructor function as well.

- Eager, but not call-by-value (evaluation of arguments is initiated whether they are needed or not).

However, note that despite Id's non-strict semantics the question of whether a particular expression will be evaluated or not (and, by extension, whether a particular side effect will occur or not) has *the same answer as it would in Scheme or ML*[5], and this is crucial to being able to reason about programs with state. One can make *local* assertions of the form "if this function is invoked, then this expression will be evaluated", or "if this predicate evaluates true, then this subexpression will be evaluated", *etc.* In lazy evaluation, on the other hand, asserting that an expression will be evaluated requires more global analysis.

The semantics described informally above have a precise description and may be found in [1, 14], but we give a flavor of it here. In a program where `f` is defined as:

```
def f x1 x2 ... xn = eBody ;
```

the application `(f e1 e2 ... en)` reduces to the parallel block:

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

(with some renaming to avoid name clashes), all of whose expressions may be reduced in parallel. A binding:

```
y1 = { x1 = e1 ;
       ...
       xN = eN ;
     In
       eBody } ;
```

reduces to a set of bindings:

```
y1 = eBody ;
x1 = e1 ;
...
xN = eN ;
```

This illustrates how a function is invoked in parallel with the evaluation of its arguments, and how it may return a result before any arguments are evaluated.

Id's rewrite rules also take a precise position on the question of *sharing* of computation (and, by extension, of 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 weak head normal form (this ensures that expressions do not get duplicated), and data structures are always referred to *via* labels, which may be regarded as abstract pointers.

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

We extend the functional language with two new operators for reading and writing the components of a data structure. Although the specific syntax depends on the type of data structure (array, algebraic type, or record), we generically call them *take* and *put* operations.

Take and put operations are *implicitly synchronized*, *i.e.*, there is no separate notion of locks or semaphores. Every data structure location is either in an "empty" state (or "unevaluated", or $\perp$) or in a "full" state, in which case it contains a value. A *take* operation on an empty slot suspends, whereas a *take* on a full slot returns the value, and resets the slot to the empty state. A *put* operation may only be applied to an empty slot.[6] If there are no suspended *take*'s, it simply writes the value there and marks it full. If there are suspended *take*'s, the slot remains empty, and the value is communicated to one of the suspended *take*'s.

For example the statement (which may be mixed with bindings in a block):

[5] Or, for that matter, in any conventional imperative language such as Fortran, C, or Lisp.

[6] 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.

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

indicates that the j'th slot of array A is to be taken, incremented by one, and the result put back. Note that the addition operator is strict, and therefore 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. Given this precedence, the semantics of takes and puts 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.

Explicit Sequencing

Occasionally, it is necessary to introduce explicit sequencing. For example, suppose we wish to reset an array location to 0 and return its old value, we might say:

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

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 *hyperstrict* in the statements that precede the barrier, *i.e.*, not only should all expressions before the barrier be evaluated but, recursively, any expressions that they may invoke dynamically (*e.g.*, due to procedure calls) should also be completely evaluated.

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).

2.4 M-structures in Algebraic Types

Here is a type declaration for a polymorphic list ("*0" is a type variable) where each tail is updateable, indicated by "!":

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

Updateable fields are called "m-fields".

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 integer above, are selected using "`x.MCons_1`". The value of an m-field may be *taken* using the notation "`x!MCons_2`". A new value v may be *put* into an empty m-field using the statement (which may be mixed with bindings in a block):

```
x!MCons_2 = v
```

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

Consider the problem of insert an integer `x` into a set `ys`, taking care not to insert duplicates. Represent the set as a list of integers, here is a purely functional solution using pattern matching:

```
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) ;
```

This function is rewritten without pattern matching as follows:

```
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) ;
```

Here is a solution using M-structures:

```
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.

In our graph traversal example in Section 4, we will use tables to record the identifiers of graph nodes that we have already visited so far, in order to avoid repeated traversals of shared subgraphs and cycles. There, we frequently need to test if an identifier is already in the table and, if not, to insert it into the table. In a parallel language, these two actions must be done *atomically* since two concurrent processes may test `x` for membership in the table simultaneously, and only one should find that the node has not yet been visited.

We modify our `insert` function above 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) } ;
```

Again, note that it has the same pipeline parallelism as our previous `insert` functions.

Finally, we show the implementation of a parallel hash table, which we will use in our graph traversal example in Section 4. 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} ;
```

with each bucket of the array initialized to an empty `mlist`.

Here is a function to test whether an integer `x` is in the hash table and, if not, insert it as part of the same atomic action:

```
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 use our previous function to test if `x` is in that set and to insert it if not. We *put* back the (possibly) new set and return the boolean result.

The 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.

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.

2.5 Our Experiments

To determine the validity of our claims, we programmed several examples (described in the next two sections) in Id using approaches A and C. Id programs compile into dataflow graphs, which constitute a parallel machine language (a partial order on instructions). We measured the performance of these programs on two dataflow platforms. The first, GITA, is a dataflow simulator instrumented to take "idealized" performance measurements.[7] This second is Monsoon, a single, 64-bit, 10MIPS dataflow processor prototype with an attached, 4 Mword I-structure memory unit.[8] The dataflow graphs and execution platforms respect the non-strict semantics and parallel evaluation order describe earlier in this section.

We evaluated the programs along three dimensions:

1. **Declarativeness:** We give a subjective analysis of how closely the program reflects the constraints of the problem. Programs which add constraints, *e.g.*, unnecessary serialization, are considered less declarative.

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

[8] These processors are being built in partnership with Motorola, Inc. Sixteen processor versions of Monsoon are expected to be available in April, 1991.

2. **Instruction Counts and Storage Use:** Instruction counts measure the efficiency of the solution as the total amount of work required to compute the answer. Instruction counts are affected significantly by storage use. Programs that do a lot of storage copying not only use a lot of store, but also pay in instruction counts for allocation, copying and garbage collection. Further, excessive storage requirements limit the size of data on which programs may be run.

3. **Parallelism:** The average parallelism of the program is computed by dividing the instruction count by the "critical path", *i.e.*, the time (in instruction steps) required to return an answer.

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 "inherent" parallelism of a program. The parallelism exhibited under idealized execution is an upper bound on achievable parallelism. Thus, algorithms with more inherent parallelism should be better able to capitalize on emerging parallel hardware (such as Monsoon) for scalable increases in performance.

A major idealization in our simulator is that each heap allocation is assumed to take one instruction. In fact, since all the slots of an object must be initialized, at least for garbage collection reasons, the cost should be proportional to the size of the object. In a parallel system, where heap allocation is tied with load balancing, the cost is likely to be even higher.

3 Histogramming a tree of samples

Suppose we are given a tree T of numbers, where each number is in the range 1 to 5, and we wish to build a histogram of the numbers, *i.e.*, an array H from 1 to 5, such that H_i = the number of times i appears in the tree. Here is the type definition for the tree:

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

This problem statement is a highly abstracted account of MCNP, a real application program used by over 350 organizations in diverse industries such as oil exploration, automobiles, medical imaging[9], *etc.* 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. (Note that no data structure is required for the "tree"; it is simply represented by the call/return tree.) When a particle finally dies, some of its properties are collected in various histograms (*e.g.*, its final position, energy, *etc.*) This program is one that 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.

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

3.1 HA1: A Functional Solution

The program builds an initial array H0 containing 0 everywhere, and then performs a right-to-left traversal of the tree. At each leaf, where i is the contents of the leaf , it "increments" the i'th location of the array, *i.e.,* it builds a new array which is a copy of the old one 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) ;

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

Critique of HA1

Threading: In principle, the increments can be done in any order. However, the "threading" of the array through the traversal serializes the accumulation. Having to pick a particular order is an unnecessary overspecification.

Note that the threading is even worse in the original MCNP problem. There, the tree is never constructed, but rather the results and arguments are passed between recursive calls. This threading required by functional programming obscures independent nature of the accumulations.

Storage: The program builds n copies of the array, where n is the number of leaves in the tree. One could argue that single-threadedness analysis, or abstract reference counting could predict that all the updates could occur in place. But these analyses assume a *sequential* order of execution, and so will preclude parallel execution. Note also that n copies of the array translate into $5n$ loads and stores, while the problem only requires n loads and stores (one load and store per increment).

3.2 HA2: A Functional Solution using Accumulators

In this example, we build an accumulator array. Each element is initially zero. Then we call `traverse` on the tree to build a list of its leaves, and then we accumulate into the array using this list. 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 of HA2

Threading: This program clearly describes the accumulation of leaf values into the result. Traverse produces a list which is "piped" into the accumulator. The only threading comes from constructing this list.

Storage: Although an improvement, this solution requires the intermediate list for accumulation. It seems unlikely that the compiler can eliminate this list, since it requires more global analysis to thread the accumulator through the generator function.

3.3 HC: An M-structure Solution

We create an initial M-array containing zeroes everywhere. We pass it into `traverse`, which recursively passes it down to all leaves in parallel. At each leaf (containing i), we *take* the i'th count, increment it, and *put* it back, thus guaranteeing that the increment is atomic. Before returning the array, we wait at the sequential barrier. Note: this is no more strict than the functional solution 2, which had to wait until the last accumulation was done before returning the array (recall that Id barriers are reentrant and local to a block).

```
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 } ;
```

Critique of HC

Threading: Traverse passes the M-array directly to the leaves; there is no threading from one leaf to the next. Thus, the ordering not overspecified. The required synchronization, between increment operations on the array elements, are *locally* specified by the take and put operations.

Storage: Only the result M-array is constructed, and it performs the minimal number of loads and stores (puts and takes) needed for this computation.

3.4 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	GITA				Monsoon
	Total instructions executed	Critical path	Avg. parallelism	Heap store used (words)	Total cycles executed
HA1	199,757	1,452	137.6	9,250	1,326,158
HA2	72,847	7,324	9.9	2,100	1,513,746
HC	60,496	589	102.7	9	291,682

Observations

Declarativeness: We believe the Approach C solution is more declarative than the functional solutions, since it does not overspecify accumulation order by threading.

Instruction count: HA1 has significantly more instructions than the other two solutions, due to copying. This is dramatically improved by the use of accumulators in HA2. HC improves on HA2 by eliminating the intermediate list.

Heap Store Used: Copying intermediate data structures in the two functional solutions dominate their storage profiles. Imperative updates, as used in the Approach C solution, allowed storage use to be reduced by three orders of magnitude. Note that our idealized execution model hides much of the true cost of heap allocation. The simulator counts each heap allocation as a single instruction, *i.e.,* it does not charge for the cost of initializing all the locations of the heap object. This cost surfaces in the data from Monsoon. As expected, there is a significant expansion in cycle time as compared to idealized execution, due to storage allocation, function call overhead, etc. However, the *ratio* of the functional solutions to the M-structure solution also grows. For example, the ratio $\frac{HA1}{HC}$ is 3.3 for idealized instruction counts, but the 4.5 for Monsoon cycles. Further, in a real parallel implementation, heap allocation can be even more expensive as it has to take into account load balancing. Thus, the comparative performance of the functional solutions are likely to worsen on a real multiprocessor.

Parallelism: Although HA1 has more parallelism than the other two, this is mostly due to copying overhead. This copying cost grows with the number of "buckets" in the histogram. Although HA2 has some initial parallelism, the algorithm is sequentialized by the intermediate list used in the accumulator. This sequentialization can be seen on Monsoon, where HA2 takes longer to finish than HA1. On a single Monsoon processor, 8-fold instruction-level parallelism is required to fully utilize the processor pipeline. The serialization in the accumulator causes poor utilization in HA2: even though it executes fewer instructions, it takes longer to complete. HC demonstrates significant parallelism, limited primarily by the sequential increments on the independent buckets.

Three other functional solutions were explored to improve performance.

- 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 produced the sum for the whole tree. Although this improved average parallelism (to 1367.5), it doubled the storage and instruction counts of HA1.
- The program performed 5 tree traversals in parallel. Each traversal itself traversed the tree in parallel, with the j'th traversal counting the number of j's at the leaves. Finally, we build the array containing the five sums. Although this had excellent average parallelism (1365) and storage use (9), instruction count suffered (266,223), because it traversed the entire tree once for each bucket.
- The program was a modification of HA1, replacing the histogram array by a balanced tree. This reduced memory and instruction overhead by reducing copying to a single path of the tree ($O(\log b)$ instead of $O(b)$, where b is the number of buckets) However, the average parallelism was dramatically reduced (to 10) due to the serialization of all updates through the root of the tree.

Although the first two solutions are attractive for parallelism, reducing memory usage (for example, by single-threadedness analysis) is hampered by their non-linear (unthreaded) structure.

4 A 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.

Given A, a node in such a graph structure, we wish to compute `rsum(`A`)`, the sum of the integer information in all nodes reachable from A.

A more complicated version of this problem would be: instead of computing the sum, build a histogram of the integers. However, we'll stick to the simple version.

Before we look at solutions to this problem, we would like to explain why we introduced unique identifiers in graph nodes. Graphs (as opposed to trees) can have shared substructures and cycles. Graph traversals need to take this into account— they must repeatedly test whether two names for a node refer to the *same* node.

In languages with side effects, the *language semantics* must take a precise position on sharing of objects in the heap. Sharing and side-effects are intimately related— one does not make sense without the other. Two names `x` and `y` refer to the same object if and only if a side-effect to the object `x` is visible as a change in the object `y`. Once a precise position has been taken on sharing in the heap, internal object addresses may be used as unique identifiers for objects. Thus, two names refer to the same object if (internally) they point at the same address in the heap. This equality test is called `same?` in Id and `eq?` in Scheme.

In pure functional languages, since there are no updates, sharing in the heap is an implementation detail and not a part of the language semantics. Thus, the concept of "sameness" must be encoded explicitly. We do this by associating, with each graph node, a unique identifier, and we test for "sameness" by testing for equality of these unique identifiers.

For uniformity, we will use the unique ids to test for node equality in both the functional and the M-structure solutions.

4.1 GA1: A Functional Solution

In order to count shared sub-graphs only once (and to take care of cycles), we need to maintain a `visited` table which keeps track of nodes that have already been visited earlier in the traversal. 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)) ;
```

Here is the code to compute the reachable sum:

```
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 } ;
```

Critique of GA1

Threading: 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).[10]

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.

Storage: Every `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 whether the tree turns out to be balanced or not. Of course, we could change the tree implementation to include rebalancing operations, but that would require an additional cost.

4.2 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 can express this easily:

[10] 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", so that later-arriving processes will not duplicate the work.

```
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 fields of nodes are ignored.

Critique of GC1

The solution is very clear. The only subtle point is the sequentialization in the `marked?` function. The reason is this: since there is no data dependency from the value that is *taken* to the value that is *put* back, we may attempt to put the constant `True` back before we have taken the old value out, and this would be an error. The sequentialization prevents this.

While this solution corresponds to most textbook algorithms, it has two drawbacks relative to the functional solution, having to do with mark initialization and multiple traversals.

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. This is unsatisfactory for two reasons:

- We may not have such independent access to the nodes, and
- The nodes reachable from the given node A may only be a small subset of the nodes in the graph, *i.e.*, it would be too expensive to reset the mark fields of *all* nodes in the graph.

Multiple traversals: Suppose we were given two nodes A and B and want to compute the reachable sum from each of them, in parallel. We cannot use the above function, as nodes marked by the traversal from A will not be counted by the traversal from B, and vice versa.

The above drawbacks are not fatal— many applications do not need this level of generality. In the next two subsections, we will see programs that overcome these drawbacks by reverting to a `visited` table to keep track of visited nodes; maintaining this table will have some additional cost.

However, here is an extension of the "mark field" idea to overcome the drawbacks. Let each traversal (each top-level call to `rsum`) be given an additional parameter, a unique *traversal identifier*, which is passed into `marked?`:

```
def rsum nd trvid = if (marked? nd trvid) then
                      ...
```

The type of the mark field is declared to be a list of traversal ids. When a node is originally constructed, the mark field is initialized to the empty list. We now define `marked?` as follows:

```
def marked? nd trvid =
    { trvids       = nd!GNode_4 ;
      (m,trvids')  = member?_and_insert_list  trvids  trvid ;
      nd!GNode_4   = trvids'
    In
      m } ;
```

i.e., it atomically tests if `trvid` is a member of the list and inserts it, if not. Now, multiple traversals can occur in parallel (they will have distinct traversal ids and so will not interfere), and all nodes implicitly have their marks reset by virtue of the fact that a brand new traversal id is in none of the mark sets. However, this solution has a garbage collection problem, *i.e.*, the mark set in a node grows indefinitely as more and more traversals are performed. We do not pursue this idea any further.

4.3 GC2: A Solution Allowing Multiple Traversals

We overcome the drawbacks listed above by reverting back to the use of a `visited` table to keep track of nodes already visited. We go back to the original definition of graph nodes without a mark field, repeated here for convenience:

```
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 } ;
```

We can implement the `visited` table in a number of ways; the parallel hash table in Section 2 is a good candidate.

Critique of GC2

Even though this solution uses a `visited` table like the functional program, it is shorter, simpler and clearer because it omits the threading of the `visited` table through the `aux` traversal. It is *more declarative than the functional solution* because it does not unnecessarily specify the order in which graph nodes should be visited (`map` visits all neighbors in parallel).

GC2': 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. Now, we can implement the table as an array of booleans, instead of a hash table.

```
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, we cannot exploit this special property of the unique identifiers; implementing the `visited` table as a balanced binary tree is still the best we can do.

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:

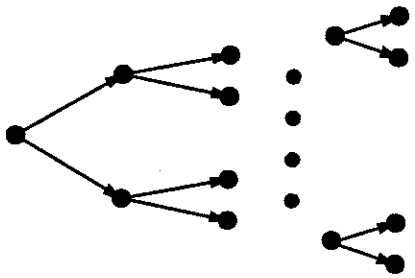
GA1	The functional program with a `visited` table implemented as an ordered binary tree.
GC2	The M-structure program with a `visited` table implemented as a hash table of size 523.
GC2'	The M-structure program with a `visited` table, implemented as an array of booleans.
GC1	The M-structure program using 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 uid generation was also used for the hash and marks version, though it does not really matter for these versions. For the "array" version, kids 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.

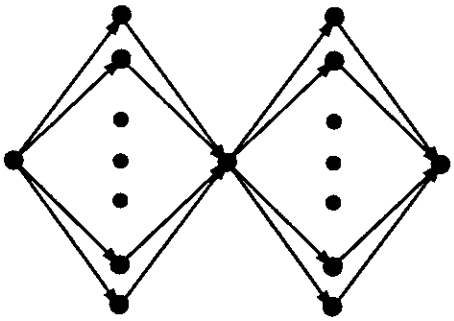


Program	Total instructions executed	Critical path	Avg. parallelism	Heap store used (words)
GA1	480,123	90,605	5.3	15,934
GC2	102,244	4,280	23.9	2,072
GC2'	55,843	2,743	20.4	524
GC1	48,002	412	116.5	0

Figure 1: Reachable Sum (rsum) on G0 (511 nodes).

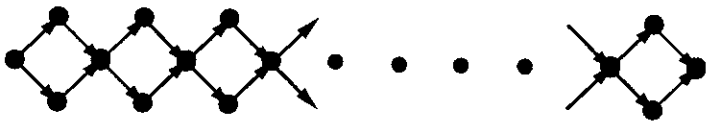
Observations

Parallelism: Graphs G0 and G1 have a high degree of parallelism, *i.e.*, there is much branching without sharing; graph G2 is not very parallel (much branching, but also much sharing), and graph G3 is also not very parallel (not much branching). Nevertheless, the functional program is unable to exploit the parallelism in G0 and G1, 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.



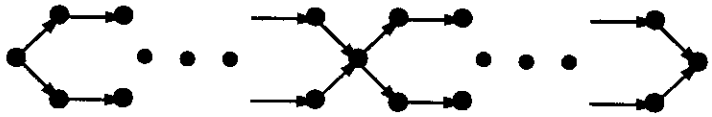
Program	Total instructions executed	Critical path	Avg. parallelism	Heap store used (words)
GA1	594,323	124,276	4.8	17,305
GC2	177,062	9,271	19.1	2,959
GC2'	95,501	9,102	10.5	524
GC1	84,078	3,728	22.6	0

Figure 2: rsum on G1 (515 nodes).



Program	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: rsum on G2 (511 nodes).



Program	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: rsum on G3 (515 nodes).

Instruction count: The instruction counts for the functional program 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). The Hash program 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 Array program takes exactly 524 words for the array of booleans. The Marks program 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 kids to keep the `visited` tree balanced.

5 Conclusions

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[16], streams[15], 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.

However, programming with M-structures becomes complicated when operations involve more than a single cell. Atomicity then 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.

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 us in 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.

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] 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.

[13] 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).

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

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

[16] 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.