LABORATORY FOR COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Supporting State-Sensitive Computation in a Dataflow System

Computation Structures Group Memo 294
March 1989

Paul S. Barth
Rishiyur S. Nikhil

This report was supported in part by the Advanced Research Projects Agency of the Department of Defense under the Office of Naval Research contract N00014-84-K-0099. Paul Barth was supported by a fellowship from Schlumberger Technology Corporation.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Supporting State-Sensitive Computation in a Dataflow System

Paul S. Barth
Rishiyur S. Nikhil

Massachusetts Institute of Technology
Laboratory for Computer Science
545 Technology Square, Cambridge, MA 02139, USA

barth@juicy-juice.lcs.mit.edu (617)-253-3219
nikhil@juicy-juice.lcs.mit.edu (617)-253-0237

Abstract

Functional languages need extensions for supporting nondeterministic, state-sensitive computation. These are required for "application" programs, such as real-time systems and database systems that respond to multiple inputs according to their temporal order. They are also necessary for "systems" programs, such as runtime support for the implementation of a functional language, which need to manipulate the state of the machine. We are interested in such computations in the presence of massive, fine-grained concurrency, such as in a dataflow system. We describe here a construct for this purpose called *managers*, resembling abstract data types. We show examples of simple managers, in which each response is immediately a function of the current input and state, as well as scheduling managers, in which the manager may choose to respond to requests out of order, for example on the basis of priority or resource availability. We show how such managers are implemented naturally in a dataflow system, and we compare our managers to various previous stream-based approaches.

1 Introduction

The universe of applications for which functional languages are appropriate continues to expand. Improvements in language design and compilation technology have allowed us to find elegant and efficient functional solutions for many applications which previously were expressed only in imperative languages. However, we are still far from being able to say that we never have to step outside the functional programming paradigm.

Some applications are inherently nondeterministic. Examples are real-time systems that respond to stimuli in the temporal order in which they occur, or a shared bank account whose response to credit, debit and balance queries depends on the prior history of transactions with other concurrent clients. Such programs demand some nondeterministic extension to a functional language.

In other situations, it may be beneficial to *use* nondeterminism because the semantics of the particular application can mask it. For example, in a simulator for a programming language, a

[†]This report was supported in part by the Advanced Research Projects Agency of the Department of Defense under the Office of Naval Research contract N00014-84-K-0099. Paul Barth was supported by a fellowship from Schlumberger Technology Corporation.

particular call to a memory allocator may return different memory blocks on different runs of the program. However, as long as the simulator does not distinguish between such blocks, the simulator may be regarded as deterministic.

Finally, there are some applications that are determinate but for which we know of no elegant or efficient solutions that are purely functional. Consider the problem of updating a node in a general graph, perhaps containing cycles (for example, in an interpreter that performs graph-reduction [12]). A functional solution to the graph-update problem is not only inefficient, because *every* path from the root to the node needs to be reconstructed, but also inelegant and nonmodular, because numerous distant, unrelated nodes may have to be reconstructed as a result of this local update.

While future research may shed more light on how to do more things elegantly and efficiently in a functional framework, we are pessimistic about functional languages being a solution to all problems. In addition, we have an immediate and pressing practical concern— the implementation of the runtime system (or *Resource Managers*) for Id [10] on the Monsoon dataflow multiprocessor [11]. These managers are used for frame allocation and deallocation, heap allocation, garbage collection, load balancing, communication with host processors, etc. In other implementations of functional languages, such resource managers are usually coded in an existing imperative language (e.g., C, assembler). An imperative language in a machine with fine-grained concurrency like Monsoon would be a nightmare to program.

Instead, we chose to establish a firm, long-term foundation for the general problem of nondeterministic, state-sensitive computation in Id. In this paper, we describe *Managers*, an experimental construct in Id akin to abstract data types, for controlled, nondeterministic, highly concurrent access to shared resources; we also describe the compilation of managers for a dataflow machine. Our approach has three broad goals:

1. **Encapsulation:** The construct should clearly delineate state-sensitive operations from pure functions. The synchronization and mutual exclusion due to concurrent access should be implicit in the construct. Further, it should be possible to express state transitions as pure functions. It should be possible to type-check managers.
2. **Expressiveness:** Managers should be first-class objects that can be dynamically created and manipulated. They should be composable to create more complex managers. They should permit nontrivial scheduling, such as responding to requests out of order.
3. **Parallelism and Efficiency:** The implementation of the construct should have a low overhead and not unnecessarily restrict parallel access to the state.

The next section describes the manager construct and gives several examples, ranging from a simple bank account to a priority printer scheduler. Section 3 describes the implementation of the manager construct on a dataflow machine; this mostly involves the design of the compilation method, but exploits some special dataflow architectural features. In Section 4 we compare our approach to previous proposals for supporting nondeterministic and state-sensitive computation in functional languages. Section 5 contains concluding remarks.

2 Managers

The Id programming construct for state-sensitive computation is the *manager*. Conceptually, a manager is a high-level interface to a *state variable*, which is shared by an arbitrary number of processes. These processes invoke state-sensitive computations through manager interface functions,

which update the state variable and return a result. The manager synchronizes incoming requests to ensure exclusive access to the state variable. Figure 1 illustrates this view of managers.

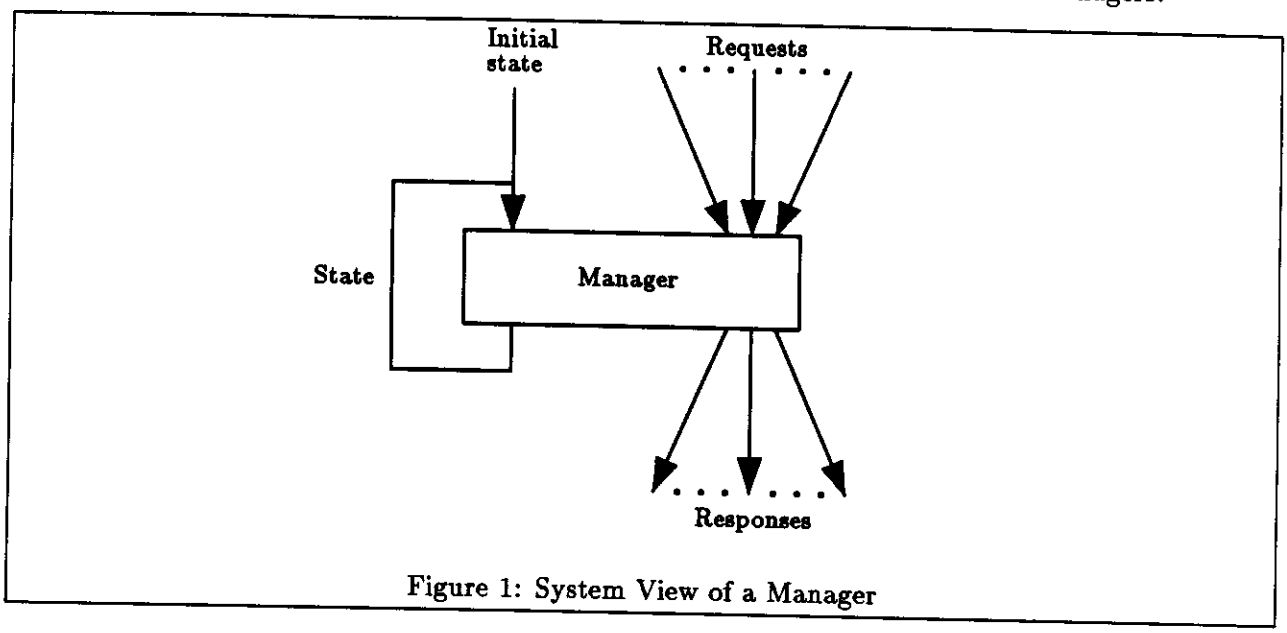

Figure 1: System View of a Manager

2.1 Encapsulating State Operations

Managers encapsulate the state variable like abstract data types. Specifically, each manager defines an abstract type for the state variable, with a set of query and update operations (called *request handlers*), and a constructor for creating instances of the type. State variables can be accessed only through this interface. In addition to defining the interface functions, the abstraction hides the synchronization and updates on the state variable, so that request handlers can be written functionally. The program interface to managers is similar to other abstract types: manager instances are created (with some initial state) and passed to functions who may make requests on the instance. Manager requests return a result, just like other function calls. The state update is entirely transparent to the caller.

As an example, consider a bank account program. An account is created with an initial deposit, against which credits and debits may be issued. If a credit is issued, the balance is updated and acknowledged with an `Ok` response. If a debit is issued, the balance is checked and updated if it is sufficient, and `Ok` is returned. If the balance is insufficient, it is not updated and `No` is returned. This example is shown in Figure 2.

As with abstract types, the manager construct has two components: the *interface* definition (the first four lines), and the *body* (everything following in the braces). The interface defines the types of the request handlers and the arguments to the constructor. These definitions make no reference to the associated state. The interface to `simple_account` consists of the three request handlers, `credit`, `debit`, and `query`, and the account constructor, called `Simple_account`. The constructor name is a capitalized version of the manager type being defined. The request handlers take an account instance and other arguments (if any) and return a response (either a balance or an acknowledgement). The constructor takes an initial balance and returns an account instance.

The manager body defines the request handlers and the initial state of the manager instance. Here, the request handlers are written as functions from the account balance and other arguments to a

```
typeof ack = Ok | No ;

MANAGER simple_account init_bal =
  typeof credit = simple_account -> num -> ack ;
  typeof debit  = simple_account -> num -> ack ;
  typeof query  = simple_account -> num
  { INITIAL_STATE = init_bal ;
    credit bal amt = bal+amt,Ok;
    debit bal amt  = if (bal >= amt) then
                       bal-amt, Ok
                     else
                       bal,No ;
    query   bal    = bal, bal
  } ;
```

Figure 2: A Simple Bank Account

pair of values: the new balance, and the response. The expression for the initial state follows the keyword INITIAL_STATE, and is evaluated when a manager instance is created using Simple_account. The manager body may also contain local values or function definitions.

The code template below illustrates the use of this manager:

```
{my_accout = Simple_account 0;

 ... (debit my_account 5) ...

 ... (credit my_account 7) ...

 ... (query my_account) ... }
```

Here, the Simple_account constructor creates the instance my_account with a balance of 0. Three requests are made on the account, a debit of 5, a credit of 7, and a balance query, which is returned. Id is an implicitly parallel language, which serializes requests based only on data dependencies. Assuming there are no dependencies between these requests, they may execute in parallel. The order in which they are processed is indeterminate, but the manager construct guarantees serializability[9]. That is, the value of the state variable following the processing of these two requests will be the same as some serial order of these requests. In this example, the result of the query request will be either 0, 7, or 2, which correspond exactly to the possible balances.

2.2 Types

The manager construct defines an abstract type T (possibly polymorphic). The arguments following the manager name are formal arguments to the the constructor, denoted by the capitalized version of the manager type. The signature of the constructor is determined by these arguments, but of course must return an object of type T. Each request handler has two type definitions, internal and external, to allow the state variable to be hidden from requestors. The external type of each handler is given its interface type declaration, of the form:

$$T \rightarrow T_1 \rightarrow ... \rightarrow T_n \rightarrow T_r \tag{1}$$

Therefore, users see each handler as a function call, taking the manager instance and other arguments and returning a result of type T_r.

A handler's internal type is similar to the external, but makes the state variable (of type T_s) explicit:

$$T_s \to T_1 \to ... \to T_n \to T_s \times T_r \tag{2}$$

That is, the arguments are identical to the interface, except for the first, where the manager is replaced with the (current) value of the state variable. The expression returns a pair, the new state and the result.

The dual types of request handlers indicate the implicit management of the state variable. The first argument is transformed from a manager in the external interface to its associated state in the body. Conversely, the pair returned by the body is partitioned into a new state and a result that is returned at the interface. Note that the first argument is the only argument to go through such a transformation: other arguments, even those referring to managers, are passed directly into the body. This allows networks of managers to be defined, as described below.

2.3 Scheduling with Managers

Managers defined thus far allow a wide range of state-sensitive programs to be written, such as data bases and priority search programs. These programs channel information between computations through the state variable. However, some programs require state-sensitive scheduling.

Scheduling reorders manager requests, deferring some while allowing later ones to proceed. This requires a facility for deferring the result of a request handler. We call this facility a *mailbox*. Every manager request has an associated return address for its response; this address can be referred to in a request handler with the keyword MAILBOX. If the response computed by a request handler is the keyword LATER, the response is deferred until a send operation is performed on the mailbox. This sends a value to the mailbox, which is then returned by the deferred request handler. For example, the body of the query handler could be written

```
query bal = { send MAILBOX bal
              In
                bal,LATER}
```

Here, the mailbox is used explicitly to return the result of the query. Typically, however, the state variable is used to store the mailboxes of deferred requests until other request handlers send them results.

For example, consider a slight variation on the bank account. In this scenario, debits greater than the current balance are deferred until sufficient credits are made. Incoming credits are applied against deferred debits, and the satisfied debits are processed. Credits and debits return acknowledgements, while queries return the balance. The manager in Figure 3 implements this example.

The interface to this account manager is identical to the previous one. However, the body is significantly different. The state variable contains two items: the current balance, and a list of deferred debits. The debit request handler shows how debits are deferred. If the balance is sufficient to cover the debit, it is processed as usual. If not, the state is updated by adding the debit's amount and mailbox to the list of deferred debits, while leaving the balance the same. The response part of the result is LATER, indicating that the response will be determined later by a send operation on the mailbox.

```
MANAGER nice_account init_bal =
  typeof credit = nice_account -> num -> ack ;
  typeof debit  = nice_account -> num -> ack ;
  typeof query  = nice_account -> num
  {
    % the state is pair: (balance,list of deferred debits)
    INITIAL_STATE = (init_bal,Nil) ;

  % Here is a local function that runs down the list of pending debits,
  % discharging whatever it can, returns a new balance and remaining pending
  % debits. It is not fair about the order of servicing debits.

    discharge_debits bal Nil          ds  = bal,ds
  | discharge_debits bal ((amt,M):ds) ds1 =
        if (bal >= amt) then
          { newbal = bal - amt ;
            send M Ok
          In
           discharge_debits newbal ds ds1 }
        else
           discharge_debits bal ds ((amt,M):ds1) ;

    credit (bal,ds) amt = (discharge_debits (bal+amt) ds Nil), Ok;
    debit  (bal,ds) amt = if (bal >= amt) then
                              (bal-amt,ds),Ok
                          else
                              (bal,(amt,MAILBOX):ds),LATER;
    query  (bal,ds)     = (bal,ds),bal
  } ;
```

Figure 3: The Nice Bank Account

Credits return an `Ok` response, and use the local function `discharge_debits`[1] to apply the credit against any outstanding debits. For each entry in the list of debits, this function tests if the balance is sufficient to cover the amount. If so, the amount is subtracted and the `Ok` acknowledgement is sent to the associated mailbox. The entry is removed from the list, and the function is called recursively on the rest of the list with the remaining balance. If the amount of the entry exceeds the balance, the entry is saved and the rest of the list is tested recursively. When the entries are exhausted, the remaining entries and balance are returned as the new state.

Mailboxes fit naturally in the manager type discipline. Recall that each request handler returns a result of type T_r. The type of the mailbox associated with a request handler is (`MAILBOX` T_r). The type of the keyword `LATER` is clearly T_r, while the signature of the `send` function is (`MAILBOX` T_r)$\rightarrow T_r \rightarrow$`void`. Id's normal type checking mechanism can therefore be applied to mailboxes.

Note the expressiveness of the manager construct when extended with mailboxes. Requests can arrive, be reordered, and processed with the addition of simple queuing operations. Although this paradigm is very flexible, it opens the door to a new class of programming errors: sending multiple values to a mailbox. Since mailboxes are first-class objects, they could be passed to several functions that perform send operations. Alternatively, a request handler could release its mailbox *and* return a result, rather than the keyword `LATER`. Also, if a mailbox is ever lost, the result of the request handler is never returned. Type-checking may uncover some of these errors at compile-time, but most can only be detected at run-time. Of course, lost mailbox errors may never be detected. The potential for error seems inextricably intertwined with scheduling flexibility: constraints on the uses of mailboxes would also constrain the class of schedulers that could be written.

2.4 Manager Networks

Individual managers can support a wide range of programs. However, many applications may require several managers working in concert. The treatment of managers as first-class objects allows networks of managers to be defined using standard data structures such as lists and arrays.

As an example, consider Dijkstra's problem of the dining philosophers [5, 14]. There are five philosophers who share a dining table. Each philosopher is assigned a seat, and requires the forks on either side to eat the meal in the center. There are five forks, one between each pair of plates. Each philosopher lives a dreary existence of alternating between thinking and eating. When a philosopher is ready to eat, he contends with his neighbors for the forks on his left and right. This contention can lead to deadlock, where each philosopher holds his left fork and is waiting for his right. A simple solution allows only four philosophers at the table at a time. In this model, there are four tickets at the center of the table. A philosopher takes a ticket, picks up his left and right forks, and eats. When finished, the philosopher returns his forks and ticket. This solution is given in Figure 4.

The managers for the forks and refectory are special cases of a limited resource, which corresponds to a semaphore.[2] Each philosopher is an infinite process that requests a ticket and access to his forks (`left` and `right` compute suitable indices). After eating, the forks are relinquished and the philosopher leaves and returns to thinking. The `THEN` construct synchronizes these actions by delaying an operation until all the previous operations have finished. The processes are initialized by calling the `philosopher` function on the seat numbers, one through five.

[1] This is a multiline function definition that uses pattern-matching to choose the appropriate clause.

[2] Although the manager definition is given here, semaphores are a degenerate case of a locked data cell, which has hardware support. Such cells (described in the next section) could be directly used here.

```
MANAGER limited_resource max =
  typeof take   = limited_resource -> ack ;
  typeof return = limited_resource -> ack
  {

    % state is a pair: (# using resource, list of mailboxes waiting for resource)
    INITIAL_STATE = (0,Nil) ;

    take (n,Ms) =  if n == max then (n,(MAILBOX:Ms)),LATER
                   else (n+1,Ms),Ack

    return (n,Nil)  = (n-1,Nil), Ack
  | return (n,M:Ms) = { send M Ack
                        In
                          (n,Ms),Ack } ;
  } ;

% An array of five forks
forks = {array (1,5)
         | [j] = limited_resource 1 || j <- 1 to 5 } ;

ticket = limited_resource 4 ;

philosopher j = {while true do
                      cogitate
                 THEN take ticket
                 THEN L = take forks[left  j] ;
                      R = take forks[right j]
                 THEN ingest L R
                 THEN return forks[left  j] ;
                      return forks[right j]
                 THEN return ticket} ;
```

Figure 4: The Dining Philosophers

2.4.1 Composition

This method does not show how to compose managers, i.e., to define a manager as a network of sub-managers. Composition supports abstraction and modularity, which is of particular importance to managers since, unlike functions, the properties of a network of managers cannot be easily determined by the properties of its members.

For example, consider a print queue in which short jobs are printed before long jobs. Requests are classified as long or short and are forwarded to a printer scheduler. The printer scheduler sends the request to the printer driver if it is free, otherwise the job is deferred. The driver prints a job, notifies the original requestor that it is finished, and sends an acknowledgement back to the scheduler. When the printer scheduler receives an acknowledgement, it sends the next job to print (giving short jobs priority), if any. Figure 5 depicts this network. The solid lines depict function and manager calls, and the dashed line depicts the final acknowledgement to the client that his job has been printed.

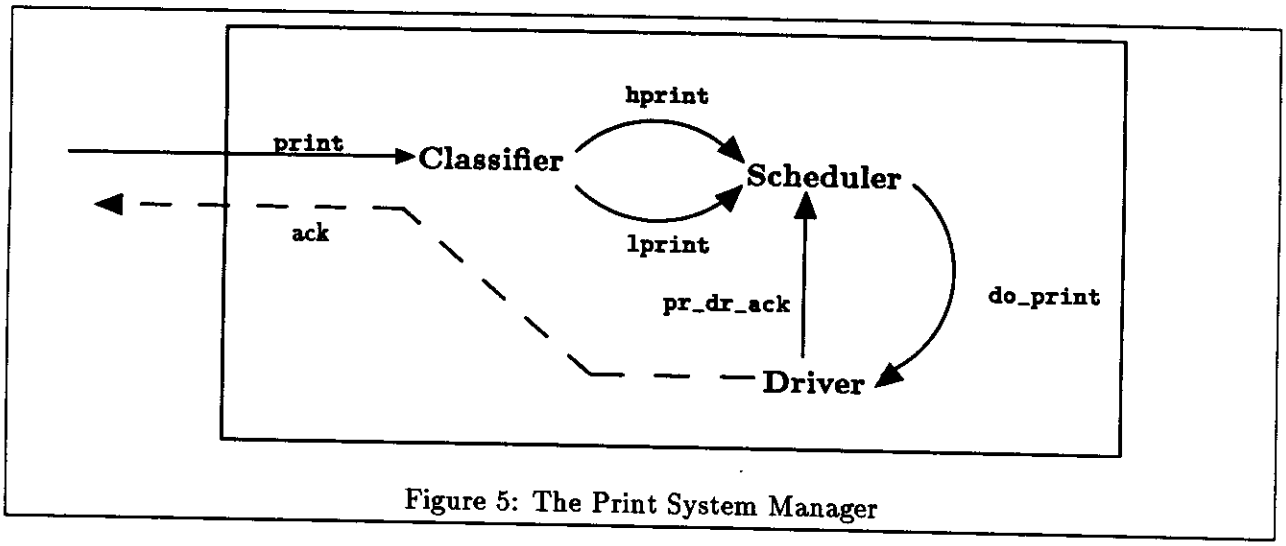

Figure 5: The Print System Manager

The print scheduler (shown in Figure 6) has three request handlers. It accepts jobs for printing *via* `hprint` (high priority) and `lprint` (low priority) requests. Each job is accompanied by the mailbox for the acknowledgement to the user. The scheduler also accepts `pr_dr_ack` messages, which are acknowledgements from the printer driver hardware that it is free and ready to accept another job. The print request handlers take a job and mailbox as arguments, and return no result (i.e., a result of type `void`). These handlers print the job if the printer is free, otherwise enter the job and mailbox on the appropriate queue.[3] The acknowledgement handler issues a `do_print` request to the driver (`pr_dr`) with the highest priority job, if any.

When given a job to print and an acknowledgement mailbox using `do_print`, it prints it on the hardware, and then sends an acknowledgement to the mailbox and an acknowledgement to the printer scheduler (`pr_sch`) indicating that it is free and ready to accept the next job.

Figure 7 shows the printer driver, which encapsulates the interface to the printing device. It accepts `do_print` requests with a job and mailbox, and prints the job. When finished, it sends two acknowledgements, one to the user's mailbox and another to the printer scheduler.

[3] List operations are used here for simplicity, resulting in LIFO queuing. The use of queue operations would allow FIFO scheduling.

```
MANAGER printer_scheduler pr_dr =
  typeof hprint    = printer_scheduler -> job -> (MAILBOX ack) -> void ;
  typeof lprint    = printer_scheduler -> job -> (MAILBOX ack) -> void ;
  typeof pr_dr_ack = printer_scheduler -> void
  {
    type printer_status = Free | Busy ;

    % state is triple:
    %       (printer status, list of deferred hi-priority jobs, list of deferred lo-priority jobs)
    INITIAL_STATE = (Free,Nil,Nil) ;

    hprint (Busy,hjs,ljs) job M = (Busy, (job,M):hjs, ljs), Void
  | hprint (Free,Nil,Nil) job M = { do_print pr_dr job M
                                  In
                                    (Busy, Nil, Nil),Void } ;

    lprint (Busy,hjs,ljs) job M = (Busy, hjs, (job,M):ljs), Void
  | lprint (Free,Nil,Nil) job M = { do_print pr_dr job M
                                  In
                                    (Busy, Nil, Nil), Void } ;

    pr_dr_ack (Busy,Nil,Nil       ) = (Free, Nil,Nil),Void
  | pr_dr_ack (Busy,(hj,M):hjs,ljs) = { do_print pr_dr hj M
                                      In
                                        (Busy, hjs,ljs),Void } ;
  | pr_dr_ack (Busy,Nil,(lj,M):ljs) = { do_print pr_dr lj M
                                      In
                                        (Busy, Nil,ljs),Void }
  } ;
```

Figure 6: The Print Scheduler

```
MANAGER printer_driver pr_sch =
  typeof do_print = printer_driver -> job -> (MAILBOX ack) -> void
  {
    % ... implementation runs hardware, but
    % ... has the following functionality
    do_print driver_state job M = { ... print job on hardware ...
                                  THEN send M Ack ;
                                       pr_dr_ack pr_sch
                                  In
                                    new_driver_state, Void }
  } ;
```

Figure 7: The Printer Driver

Finally, we have the entire print system, a composite manager that also keeps count of the total number of jobs accepted for printing. This is depicted in Figure 8.

```
MANAGER print_system =
  typeof print = print_system -> job -> ack
  { INITIAL_STATE = 0 ; %  Total num of jobs printed

    pr_sched = printer_scheduler pr_dr ;
    pr_dr  = printer_driver pr_sched ;

    classifier job M = if large? job then
                         hprint pr_sched job M
                       else
                         lprint pr_sched job M

    print n job = { classifier job MAILBOX
                  In
                    n+1, LATER }
  } ;
```

Figure 8: The Print System Composite Manager

It has a single request handler, `print`, that takes a job and returns an acknowledgement when the job has actually been printed. The network of sub-managers is defined in the body by two mutually recursive equations instantiating a `printer_scheduler` and a `printer_driver`, indicating their cyclic request structure. When the print system is instantiated, these two managers are instantiated as well. In essence, this set of definitions form a template for a network that is dynamically created for each instance.

The `print` request handler calls the `classifier` function with the job and its mailbox, increments the number of jobs received, and defers its result until the job is printed. `Classifier` sends a high or low print request to the scheduler, along with the job and the mailbox.

Note how mailboxes are used in this system. Rather than being circulated in the state of the print system, they are passed between sub-managers, and finally satisfied by the print driver. This frees the network from a strict call-return paradigm, allowing it to be structured as processes. This is natural for applications such as printers, where requests for a resource are processed in parallel with its use. Furthermore, it allows very efficient implementation, as described in the next section.

2.5 Summary

Managers meet our goal of encapsulation. The state variable and its associated operations are defined in a single construct. The external interface to a manager is clearly defined by the request handlers, which can be written as as state transition functions. Further, the state variable is completely hidden from the interface. This interface defines the scope of the nondeterminism in the program: any function calling a request handler (or another nondeterministic procedure) is potentially nondeterministic.

Managers also meet our goal of expressiveness. Mailboxes allow requests to be deferred and re-ordered with the addition of simple queuing operations. Managers can be composed into networks of asynchronous processes in an abstract and modular way. The treatment of managers and mailboxes as first-class objects increases their flexibility.

It is important to note that managers employ nonstrict execution to realize this expressiveness. Request handlers compute the new state and response independently; nonstrictness allows them to be returned asynchronously. This means that the state can be returned (and therefore relinquished for other requests) before the response is computed. Naturally, this supports scheduling, where responses are deferred until some future request. However, this generalizes to support mutually and self recursive managers. Since the state is returned immediately, the response may be determined by another request of the same manager. The state will be available to the new request, so deadlock will not result. This differs from most abstractions supporting mutual exclusion, such as monitors[8] and critical sections[4]. Of course, request handlers must not use the manager recursively while computing the new state. The mutual exclusion guarantee on the state variable would produce deadlock in this case.

Managers also meet our goal of efficiency, as described in the next section.

3 Implementation

In the runtime system that we are implementing for Id, managers are used for frequent, common activities such as allocation and deallocation of activation frames and heap objects. It is thus very important that manager implementations be extremely efficient. There are two key issues:

- ensuring mutual exclusion, i.e., two requests to the same manager must acquire the manager state serially, and
- implementation of `MAILBOXes`, `LATER`, `SENDs`, etc., for scheduling managers.

It is clear that we need some notion of a *locked cell* for mutual exclusion. Further, to avoid busy-waiting, we need some way of queuing waiting processes on locked cells and awakening a waiting process when the cell is unlocked. Also, because of fine-grained concurrency, these operations must be very efficient. All this is standard; what is novel is the way these things blend in naturally with existing dataflow concepts [3]. Our target dataflow machines already have the notion of a *tag*, which can be viewed as a process descriptor. Further, the memory system in such machines are *I-structure memories*, which are already capable of queuing a set of tags, i.e., a list of processes. We explain these concepts first, and then show how, with minor extensions, they are used to implement managers.

3.1 Dataflow Tags and Locked I-Structure Cells

Tags

In the von Neumann model of computation, there is only a single thread of computation. The instruction to be executed is specified by a single, global *Instruction Pointer*, or IP.[4] The dynamic context for that instruction is specified by a single, global *frame pointer*, or FP. Of course, the dynamic context may also include a set of registers but, at a minimum, one needs an IP and an FP.

[4] We prefer this term to the term "Program Counter".

Even a single dataflow processor is capable of supporting thousands of such threads. Each thread is described by a *token* which contains, among other things, an (IP,FP) pair. We refer to an (IP,FP) pair as a *tag*; it can also be viewed as a continuation, or as a thread or process descriptor.

Split Memory Transactions

Performing a remote memory read or write in a dataflow architecture is quite different from the corresponding actions in conventional parallel architectures, and is central to the advantage of dataflow for multiprocessors. Suppose we had an expression:

```
- (hd x)
```

The subexpression `(hd x)` must perform a memory read (say, to address `A`), whose result must be delivered to the negate instruction (-). In a conventional machine, the processor sends a `READ A` message to memory, which responds with a value (say, `v`), and execution resumes. The read is synchronous in that the processor waits for the response.

For a dataflow processor, a remote memory read is always a *split transaction*. The processor sends the memory a token (message) of the form:

`READ A (IP`$_{neg}$`,FP)`

where `IP`$_{neg}$ is the address of the waiting negate instruction and `FP` is its dynamic context. In other words, a request to read a remote location is always accompanied by the continuation that is awaiting the result. The processor can then immediately continue executing other threads. When the message is received at the remote memory unit, it responds with a message:

`(IP`$_{neg}$`,FP) v`

When this message, in turn, arrives back at the processor, execution can resume exactly at the waiting continuation with the required value. In this way, dataflow processors are very good at tolerating long latencies— they do not idle during memory reads.

I-structure Memory

The heap area of memory is implemented with I-structure memory units. Unlike ordinary memory units, these have an additional synchronization capability for implementing immutable data structures for functional languages. Every cell has extra "presence" bits that designate it as Full or Empty. Objects in the heap initially have their component cells set to the Empty state. If a `READ` token arrives at an empty cell, the accompanying (IP,FP) is simply queued at that location— the `READ` is said to be *deferred*. When the cell is finally written using a token:

```
WRITE A v
```

the value is also sent to all the deferred readers, i.e., to all the continuations queued at that location.

Figure 9 summarizes I-structure memory units, which are capable of accepting `READ`s accompanied by continuations, have presence bits at each location, and have the ability to defer `READ`s at empty locations and satisfy them later when they become full.

Note that the I-structure unit is capable of managing the space for the deferred lists locally. That is, allocation and deallocation of list cells for deferred tags is done internally within the memory unit.

Lock Operations

By observing that a deferred list of tags is simply a process queue, it is but a small step to extend an I-structure memory unit to handle lock operations. When it receives a token of the form:

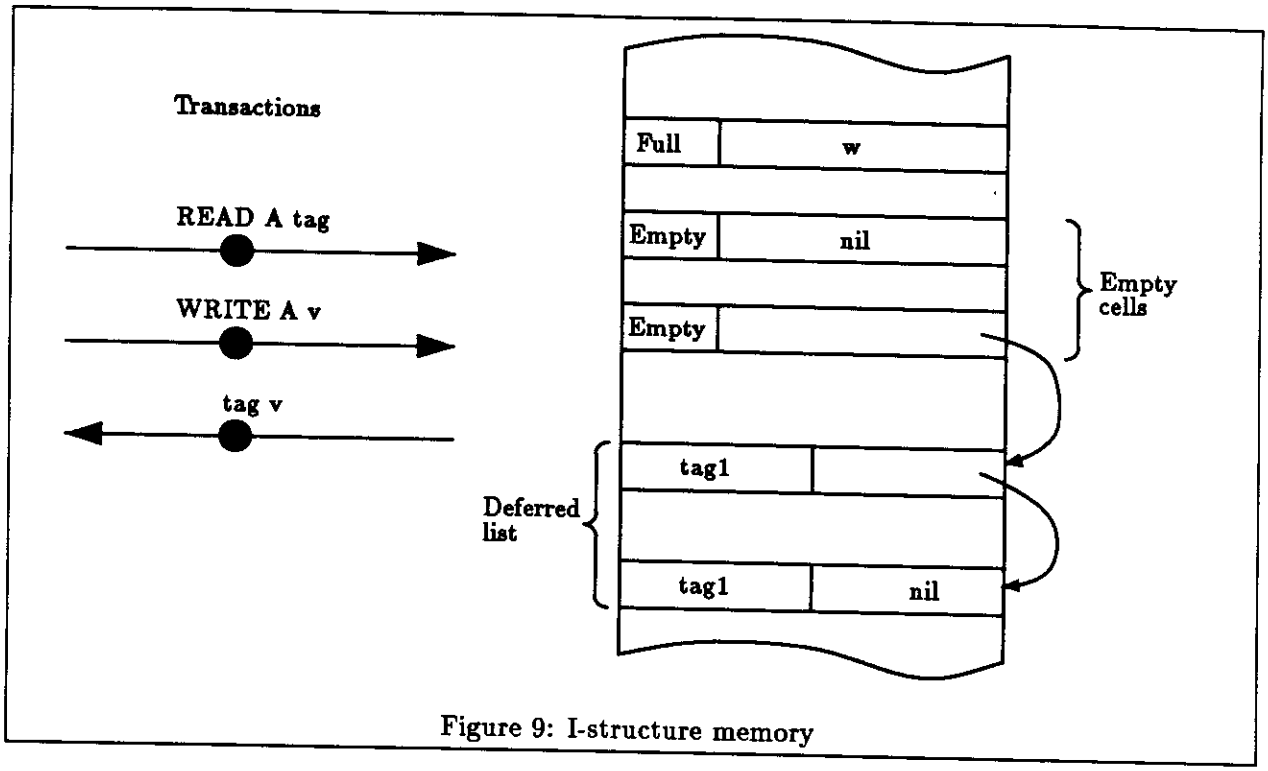

Figure 9: I-structure memory

```
READ_AND_LOCK A (IP,FP)
```

if the location A is Empty, it enqueues the tag there, as with an ordinary READ operation. When it receives a token:

```
WRITE_AND_UNLOCK A v
```

it behaves thus. If the location is Empty and there are no deferred tags, it simply writes the value there and changes it to Full. If the location is Empty and there *are* deferred tags, it removes one tag from the deferred list and sends the value v to that tag. If the location is Full, it is an error.

Note that there is no busy-waiting at all. The thread that requires the result of the READ_AND_LOCK travels to remote memory, waits there in a deferred queue, and comes back to the processor to resume executing. Meanwhile, the processor is free to continue executing other threads.

Implementing Managers and Simple Request Handlers

It should now be easy to understand the implementation of managers and simple request handlers, i.e., those that respond immediately to each request (such as the simple bank account).

At minimum, a manager is simply a single cell in I-structure memory containing the object that represents its state. In the dataflow machine, scalar objects are stored directly and non-scalar objects are represented by pointers to objects elsewhere in I-structure memory.

Consider the simple bank account manager, repeated here in outline:

```
MANAGER simple_account init_bal =
  typeof credit = simple_account -> num -> ack
  ...
  { INITIAL_STATE = init_bal ;
    credit bal amt = bal+amt, Ok;
    ...
  }
```

The constructor is implemented as follows:

```
Simple_account init_bal = (allocate_locked_cell init_bal) ;
```

The `credit` handler is implemented as follows:

```
credit cell amt = { bal = read_and_lock cell ;
                    new_state,result = bal+amt, Ok ;
                    write_and_unlock cell new_state ;
                  In
                    result } ;
```

Note that though the internal definition of the handler specifies the state as an argument, it is actually called with the manager (the pointer to the locked cell), in accordance with its external type. We fetch the current state and lock the cell, compute the new state and result, using the original handler body expression, then write back the new state and unlock the cell. If two threads invoke handlers on the same manager at the same time, only one of them succeeds at the `read_and_lock` and the other gets deferred. The winner computes the new state and, when it sends it back to the cell using `write_and_unlock`, the new state is passed on to the second thread. Thus, we achieve the goal of mutual exclusion and serialization on the state variable. Note that the implementation is a simple source-to-source transformation on the original manager definition.

There is one other complication that we need to take care of, due to the parallelism and nonstrictness of the language. In Id, all statements in a block execute in parallel, except where limited by data dependencies. If there were no data dependency between `state` and `new_state`, the `write_and_unlock` may occur *before* the `read_and_lock`, which is clearly undesirable. An example of a manager handler with this property would be a handler that resets the state to some fixed, known value. To ensure correct ordering, we make the `write_and_unlock` operation strict in both arguments, and we gate the second argument as follows:

```
write_and_unlock cell (strict_K new_state state)
```

where `strict_K` is just a version of the K combinator that is strict in both arguments.

With this implementation, the body of the manager is very efficient. For handlers like those in the simple bank account, it boils down to as few as three or four instructions. The only remaining overhead is function call overhead, which is often eliminated by the compiler using inline substitution.

The implementation also allows much parallelism. Multiple requests can run in parallel, including multiple versions of the same request handler. The locked cell provides the mutual exclusion necessary to avoid interference, but it does not serialize the state-independent portions of the handlers. For example, credits to the bank account can be acknowledged before the balance is actually available. Further, since the handler specifies a *function* from state objects to state objects, the handler can continue computing on the old state object even after it has returned the new state object to the manager cell.

The implementation is also space efficient. The storage for the simple bank account amounts to exactly one cell containing an integer (the current balance); compare this with solutions that are expressed as stream transformations.

In some managers, such as the priority printer, arguments given to the manager constructor are not only used in the initial state expression, but are also used as constants in the bodies of the handlers. In this case, the manager object is actually a tuple of cells, of which only one cell is the locked cell containing the manager state. The remaining cells contain any constructor arguments that may be referred to in the body of a handler. In the handler bodies, a reference to a constructor argument is compiled as a fetch from the appropriate slot of the manager object.

Implementing Scheduling Request Handlers

The implementation of scheduling request handlers, i.e., those using mailboxes to defer responses, is a slight variation on the normal procedure call-return mechanism.

In a conventional language, the "return continuation" for a procedure activation appears as two pieces of data in its activation frame— the dynamic link (previous frame pointer) and a return-IP (an Instruction Pointer to return to). In compiling Id, every procedure has an implicit zero'th argument containing exactly this information. Its value is the (IP,FP) tag of the instruction that awaits the result of the procedure. Normally, the value `v` of the procedure body's expression is returned to the caller using a

```
SEND tag v
```

instruction. This sends `v` to the instruction specified by `tag` which, therefore, continues in the caller.

For handlers that respond to requests out of order, we essentially do these sends explicitly. The keyword `LATER` is used to indicate that the result is not returned in the normal manner, i.e., the code for returning the value is omitted at that point. The special variable `MAILBOX` is exactly a programmatic name for the return (IP,FP) tag that is normally implicit as the zero'th argument. Finally a statement of the form

```
send  mailbox-expression value
```

compiles directly into the `SEND` instruction.

4 Comparison with previous work

Languages like Scheme [13] and ML [6] are, of course, capable of state-sensitive computation because they have explicit imperative constructs. However, these are sequential languages, and our emphasis here is on highly parallel languages, where *synchronization* is essential to control access to shared state.

In [7], Henderson describes an `interleave` "function" that nondeterministically merges two input streams into a single output stream in the temporal order in which the objects of the input stream become available. Using this, a simple bank account for two users can be written as follows:

```
resps1,resps2 = untag (process_reqs init_bal
                                    (interleave (tag 1 reqs1)
                                                (tag 2 reqs2)))
```

The `tag` function attaches a name (an integer) to each request so that its source is known even after it is merged with other requests. `Interleave` merges all requests nondeterministically into a single stream. `(Process_reqs bal)` is a simple stream mapping function from tagged-requests to tagged-responses that carries along a current balance (the state), and implements the state transition rule. `Untag` strips the tags off the stream of tagged responses, returning a pair of response streams.

The method can be elegant when the topology is simple and known statically, as in the above example. When the topology changes dynamically, then n-ary tagging, n-ary interleaving, n-ary untagging, etc., all become rapidly unmanageable. Programming in this style is often referred to as "spaghetti programming".

In [15], Stoye embeds the merge operation in the operating system. Thus, each "process" is a pure function from an input request stream to an output response stream, and has a unique name (an integer, say). When process P1 wishes to send a message to process P2, it places a request tagged with "2" in its output stream. The operating system picks it up and places it in the input stream for P2. P2, in turn, places a response tagged with "1" in its input stream, which is again transmitted by the operating system into the input stream for P1. Thus the nondeterministic merge is embedded in the operating system which serves as a giant dispatching mechanism, picking up requests from all the processes' output streams and, based on the tags, moving them into the corresponding processes' input streams.

The main problem with this approach it that it does not appear to be very modular, since *all* messages in the system are routed through a single, giant, nondeterministic switch.

Resource managers for Id were first proposed in 1977 [2] and later refined by Arvind and Brock [1]. Here, the external view of a manager was a request-to-response procedure. The internal view, on the other hand, was a stream-to-stream function. The tagging and nondeterministic merging of the requests and the untagging and separation of the responses was done implicitly in the implementation.

A problem with Arvind-Brock managers seems to be that they do not compose well. In their paper, they discuss the priority printer scheduler where in addition to the manager construct they also had to use a nondeterministic **merge** operator explicitly.

There are some general criticisms of all stream-based approaches. The first is that multiple arguments to a request must be packaged into a data structure in order that they can be inserted into the request stream. These arguments are subsequently retrieved from this data structure inside the manager. In contrast, in our managers, multiple arguments are ordinary function arguments, and are typically handled much more efficiently than a heap-based data structure (in conventional terminology, they are passed "on the stack").

A second general criticism of all stream-based approaches is a question of types and type-checking. Since all requests are merged into a single stream, they must all be of the same type (the same is true for responses). To accommodate this, it is usually necessary to define a new algebraic type containing one disjunct for each kind of request, and a new algebraic type containing one disjunct for each kind of response. Each request then has to be packed into an request-type object before the interleave and unpacked inside the request processor; the response then has to be packed into a response-type object and subsequently unpacked after untagging. This can involve a substantial overhead.

In Stoye's approach, the problem of type-checking appears to be even more serious. As in Henderson's system, all requests to a process must be of the same type (and, similarly, all responses from a process). But it is worse than that. P1 may wish to send requests to two unrelated managers P2 and P3. But since all these requests must be channeled through a single stream (P1's output stream), this implies that the types of the input streams of P2 and P3 must be the same. Similarly, all responses from P2 and P3 to P1 are channeled through P1's input stream, which implies that the types of the output streams of P2 and P3 must be the same.

In our manager construct, on the other hand, each kind of request is handled by its own, distinct handler; thus, each handler can have its own type, unrelated to the types of other handlers.

A third general criticism of stream-based approaches is this. Since all requests arrive on the same stream, it necessitates a run-time case analysis and dispatch to different handlers after examining the type of a request. In our manager construct, this dispatch is done at compile time, since the programmer directly calls the appropriate handler function for different types of requests. Because this dispatch is done at compile time, calls to our managers are also amenable to compile-time optimization, the most important one being inlining.

In our manager construct, the "tagging" and "untagging" operations are done implicitly using the underlying continuations (dataflow tags) that must exist already in any implementation (whether dataflow or not). Thus, the programmer never has to worry about explicitly managing a separate namespace for processes. This makes it much easier to deal with managers as first class objects, i.e., they are named and manipulated like any other object in the language.

For simple managers (responding in the order of requests), when requests are sufficiently infrequent, there is no stream construction at all, either implicitly or explicitly. Requests are queued only when they contend for the state variable. It is not clear what compiler prowess would be needed to optimize stream-based managers to reach this level of efficiency. For scheduling managers, we explicitly build a queue of deferred requests in the handler bodies and embed it in the state of the manager; however, this is inevitable in any manager that does scheduling, stream-based or otherwise.

5 Discussion

5.1 Formal properties of programs

We introduced managers as an experimental extension to the functional language Id in order to express nondeterministic and state-sensitive computation. However, the addition of this construct comes at a price. First, we lose referential transparency and, consequently, the valuable ability to employ equational reasoning. However, this may be mitigated somewhat by the fact that managers are distinguished by syntax and by type. It should therefore be possible, through analysis, to partition a program into its functional and nonfunctional parts, and to continue to be able to reason about the functional part. We need to develop tools to reason about the nonfunctional part.

A further property that we lose is determinacy (the Church-Rosser property).[5] For applications that are inherently nondeterministic (such as a real-time signal processor) this loss is inevitable, and the best we can do is to design the programming construct to simplify reasoning about anomalous behavior such as starvation or deadlock. Again, we feel that the simplicity of the manager construct should facilitate this.

Other programs use nondeterminism but can be regarded as determinate because of the special semantics of the application. An example, cited earlier, is a memory allocator that nondeterministically returns a memory block, embedded in a program that does not depend on the specific memory block allocated. In such programs, one often wishes to establish gross properties about permutations of calls to the manager. Such reasoning may be facilitated by the fact that

[5] Note that one can lose referential transparency without losing determinacy. The I-structure constructs in Id [3] are an example of this.

we often specify request handlers as pure functions from (state,input) to (state,output), so that properties of permutations of manager calls can be tied to the question of identifying the conditions under which function composition is commutative.

Still, much work remains to be done in studying how to reason about programs with managers.

5.2 Synchronization and Strictness

A complication introduced by state-sensitive computation in the presence of nonstrictness is the need for additional synchronization constructs. Consider a call to a request handler:

```
handler mgr arg1 ... argN
```

The manager argument (`mgr`) may be available long before any of the other arguments. If the handler procedure were to lock the manager cell immediately, it would prevent other managers from accessing the cell even though it would not be doing any useful work itself until it received some or all of its other arguments. To achieve this, Turner [16] has proposed that nondeterministic procedures be *hyperstrict*. We believe that this is too stringent, and may impair concurrency unnecessarily.

Within the implementation of a handler, we have already seen that if the new state does not depend on the old state, we need explicit sequentialization (using the `strict_K` function) to ensure that the old state is retrieved before the new state is stored. Again, forcing complete sequentialization can impair concurrency, because the handler could be doing useful work computing the new state before it has completed the retrieval of the old state.

Synchronization is also of importance in composite managers. For example, in the Dining Philosophers problem, the `philosopher` function required the `THEN` construct to sequentialize its requests to disjoint managers.[6]

This fine control of strictness, for correctness without compromising concurrency, is a topic for future research.

5.3 Conclusion

Support for nondeterminism and state-sensitive computation is critical to the success and widespread use of functional languages. Managers are a construct for writing such programs that provide encapsulation, expressiveness, and efficiency. They allow for the succinct expression of concurrent state-sensitive computations, yet provide control and flexibility in their use. They have been used for a variety of applications, and will provide the foundation for analysis of programs incorporating state-sensitive computation.

Acknowledgements: The authors are indebted to the members of the Computation Structures Group at M.I.T. for their technical comments and suggestions. Arvind has been instrumental in the inception and development of these concepts. Richard Soley and Ken Steele developed locked storage cells, which are instrumental to efficient implementation. Numerous discussions with Ken Traub, Steve Heller, Jonathan Young, Jamey Hicks, Steven Brobst, Greg Papadopoulos, and David Culler significantly improved these ideas.

[6] To join the forks, as it were (pardon the pun).

References

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

[2] 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.

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

[4] P. Brinch Hansen. *Operating Systems Principles.* Prentice-Hall, Englewood Cliffs, NJ, 1973.

[5] E. W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1:115–138, 1971.

[6] R. W. Harper, R. Milner, and M. Tofte. The definition of standard ml; version 2. Technical Report ECS-LFCS-88-62, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, The King's Buildings, Edinburgh EH9 3JZ, Scotland, August 1988. Available from D.B.MacQueen, AT&T Bell Labs, Murray Hill, NJ.

[7] P. Henderson. *Purely Functional Operating Systems*, pages 177–192. Cambridge University Press, Cambridge, England, 1982.

[8] C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 10(10):549–557, October 1974.

[9] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocessor programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

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

[11] G. M. Papadopoulos. *Implementation of a General-Purpose Dataflow Multiprocessor.* PhD thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, August 1988.

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

[13] J. Rees and W. Clinger (eds.). Revised^3 Report on the Algorithmic Language Scheme. Technical report, Massachusetts Institute of Technology Artificial Intelligence Laboratory, Cambridge, MA, 1986.

[14] G. Ringwood. Parlog86 and the Dining Logicians. *Communications of the ACM*, 31(1):10–25, January 1988.

[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.