Massachusetts Institute of Technology

Laboratory for Computer Science

Computation Structures Group Memo 217

June 1982

Streams and Managers

by

Arvind
J. Dean Brock

To appear in the *Proceedings of the 14th IBM Computer Science Symposium* being published as a volume of the *Lecture Notes in Computer Science*.

This research was supported by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-75-C-0661 and by the Department of Energy under contract DE-AC02-79ER10473.

Streams and Managers

Arvind
J. Dean Brock

Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139
U. S. A.

Abstract

The sole effect of expression evaluation in a functional programming language is the production of a resultant value. This absence of side–effects greatly facilitates both the formal characterization and the concurrent execution of functional programs. Unfortunately, the absence of side–effects also conflicts with conventional means of achieving input/output, inter–process communication, and resource allocation. By incorporating the history of communication into a *stream*, functional programs can be written for I/O and communication. Using the stream concept, *managers* may be written to control access to resources shared by several processes.

1. I/O and Side–effects

In a side–effect free language, disjoint expressions may be evaluated concurrently without danger of interference. This freedom from interference, or locality of effect, is largely responsible for the parallelism obtained when applicative languages are evaluated using the dataflow graph model. Purists may differentiate between many classes of side–effect free languages: applicative, functional, dataflow, etc. We use the terms interchangeably. The important concept is the absence of side–effects.

Although many of the side–effects considered "necessary" by most programmers are easily foregone [1] (for example, by creating updated arrays instead of modifying arrays), one very important type of side effect, namely I/O, seems truly necessary. Consider a program to manage a video terminal. Such a program, among other things, echoes input characters for display on the screen and responds to the *delete* key by sending signals to reposition the cursor on the screen and erase the last input character. In a conventional non–functional language the input and output for the program is represented by files, which are explicitly mutated (changed) by the reading and writing of characters. In addition the program contains state, generally a buffer of the most recently received characters. An attempt to write the video terminal handler in a functional language immediately poses problems because without side–effects terminal interaction seems impossible to describe: two successive *delete*'s may have different effects. Also, the handler cannot be viewed as a function all of whose inputs are available before any answers are produced. Thus it would appear that I/O is anathema to functional languages. However, by changing our viewpoint of I/O, the "conflict" is resolved. In functional languages the I/O *channel* must be viewed as a special data value produced, or examined, by programs, not as a medium to be written, or read, by programs. The special I/O data value is

* This research was supported by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014–75–C–0661 and by the Department of Energy under contract DE–AC02–79ER10473.

the *stream*, a sequence consisting of all the individual items written to, or read from, the channel. From this viewpoint, a terminal is a function whose input and output are streams of characters. Other elementary I/O devices, such as printers, may be similarly represented.

Streams, introduced in Section 2, solve a number of problems associated with input/output in a functional language context. Streams can be used to write functional programs that appear to have local state: the response of a program may depend on the order in which data items are input. However, functional languages, even with streams, lack the power to express indeterminate computation such as programs to serve several video terminals, and in Section 3 we introduce *merge*, an indeterminate stream operator. Unfortunately, certain classes of problems, specifically resource allocation among several competing processes, cannot be solved with *merge*'s alone, and in Section 4 the *manager* construct is developed for this purpose.

2. Streams

Weng [19] introduced the data structure *stream* into dataflow languages along with four primitive operators *first*, *rest*, *cons*, and *empty* for manipulating them. These operators were chosen to resemble the list operators of LISP. The operation *first* yields the first element of the stream, while *rest* yields the remainder. The operation *cons*, when applied to an elementary value and a stream, produces the stream consisting of the elementary value followed by the argument stream. Thus, for non–empty streams S, S equals *cons*(*first*(S), *rest*(S)). The emptiness of streams may be tested with the remaining stream operator *empty*. The functionality of these four operators is summarized below:

first([ ]) = *error*
first([x1, ...]) = x1

rest([ ]) = *error*
rest([x1, x2, ...]) = [x2, ...]

cons(x, [x1, ...]) = [x, x1, ...]

empty([ ]) = *true*
empty([x1, ...]) = *false*

Weng's language had three major constructs for writing stream programs: acyclic blocks, conditional expressions, and recursive procedures.

It is important to note that the stream operators are *non–strict*, that is, they can, and must, be applied to their stream arguments incrementally before the entire argument stream has been computed. For example, *first*(S) can produce its result as soon as the first element of S is available — it need not wait until the computation of S is completed. The key to non–strictness is the implementation of the stream constructor *cons*. The output of *cons*(x, S) must be produced incrementally so that x, the first element of its result, may be accessed before the entire resultant stream has been computed. Weng chose the obvious incremental implementation of streams by representing them as a series of tokens passed through the arcs of a dataflow graph. Weng's implementation limits stream elements to simple (*i. e.* non–stream) value. Other implementations in which streams are represented by pointers into a "structure memory" are possible. It is

not difficult to make such an implementation non-strict. A *cons* cell can be allocated and "used" before its components are available, if the components are tagged to indicate whether or not they have yet been computed. Within the structure memory, the elements of the stream may themselves be pointers to other streams, thus yielding a more general implementation of streams than possible with Weng's original proposal.

With these LISP-like stream operators, recursion is the natural programming style. Consider the problem of writing an elementary terminal line editor EditLine which collects the characters produced by a terminal into lines. Abstractly, EditLine may be considered a function which maps the elements of its domain, the streams of characters, into elements of its range, the streams of strings. For example:

EditLine(['I', 'd', 'EOL', 'I', 'd', 's', 'y', 's', 'EOL']) = ["Id", "Idsys"],
where 'EOL' is the end-of-line character.

Using recursion, the function EditLine is written in an Weng-like language, having the syntax of Id, the dataflow programming language developed by Arvind, Gostelow, and Plouffe [3], as:

```
procedure EditLine (InStream)
  ( BuildLines(InStream, "") )

procedure BuildLines (InStream, LineBuff)
  ( if empty(InStream) then
      []
    else
      ( let CharIn ← first(InStream) ;
            RestIn ← rest(InStream) ;
        in
          if CharIn = 'EOL' then
            cons( LineBuff, BuildLines(RestIn, "") )
          else
            BuildLines( RestIn, string-append(LineBuff, CharIn) ) ))
```

The "work" of this program is done by the procedure BuildLines. First, BuildLines tests for the termination of its input stream. If more characters remain to be processed, it gets the next (i.e., *first*) stream element and tests whether or not it is the end-of-line character. If it is end-of-line, the line buffer is *cons*'ed onto the beginning of the output stream and BuildLines is recursively invoked with a new, empty line buffer; otherwise, the next character is appended to the line buffer and passed on to the next invocation of BuildLines. These programs illustrate well the most common form of recursive stream computation. On each step of the recursion, the next input value is obtained by use of *first* and the next output, if any, is *cons*'ed onto the front of an the stream produced by recursively processing the *rest* of the input. Here, the variable LineBuffer serves as the state for procedure BuildLines and is passed to all the stages of the recursion.

Id supports an iterative stream processing construct in addition to the four elementary stream operators. The iterative construct for reading streams is the *for-each* loop. On each iteration the loop header clause "*for each* x *in* S" causes the identifier x to be bound to the next element of the stream S. The iterative construct for producing streams is the *return-all* loop trailer. If "*return all* x" ends an Id loop, an output stream is generated which contains the values of x on each iteration as its elements. [Incidentally, arbitrary Id expressions may follow the *return-all*.] One final, and very useful, embellishment of the *return-all* is the *but* clause. The *but* clause may be used to disable the *return-all* for certain stream values, and thus may be used

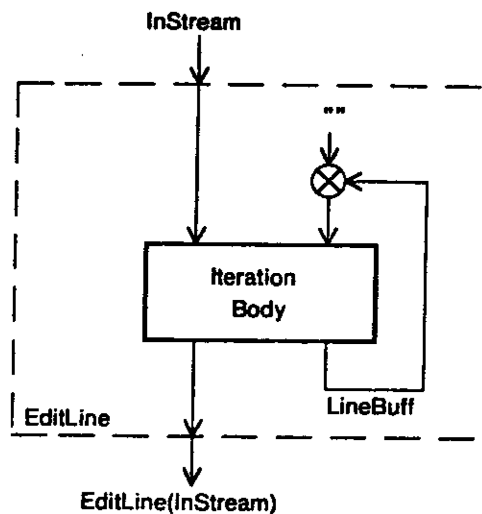
to "filter" streams. If the clause "*return all* x *but* a" follows a loop, the output stream will contain all the values of x *except* those with value a. Generally, the iterative specification of stream functions is much more succinct than the recursive specification. For example, the procedure EditLine (illustrated in Figure 1) may be written as:

```
procedure EditLine (InStream)
   ( initial LineBuff ← ""
    for each CharIn in InStream do
      new LineBuff, LineOut ←
        ( if CharIn = 'EOL' then
            "", LineBuff
        else
            string-append(LineBuff, CharIn), λ)
   return all LineOut but λ)
```

Note how the *but* clause ensures that lines are placed on the output stream only when the next input character is 'EOL'. In all other instances LineOut is bound to the value λ and then removed from the output stream.

Although we have discussed streams only in the context of dataflow languages, they have proven to be useful in other contexts, even in the formal description of non-applicative languages. In 1965 Landin [16] defined a lambda calculus based semantics for ALGOL-60 by representing the values of a loop variable as a stream. This was probably the first use of a non-strict data structure. Later, Burge [6] incorporated these ideas into the design of a functional language. In 1974 Kahn [13] recognized that the input and output channels of processes written with GET and PUT, the conventional I/O primitives for which side-effects are so important, can be viewed as streams and that, when processes are viewed as stream functions, the semantics of networks of parallel processes may be derived using a simple fixpoint theory. Friedman and Wise [9] have suggested making the list constructor *cons* of LISP non-strict. This makes LISP lists an even

Figure 1.



Iterative implementation of EditLine

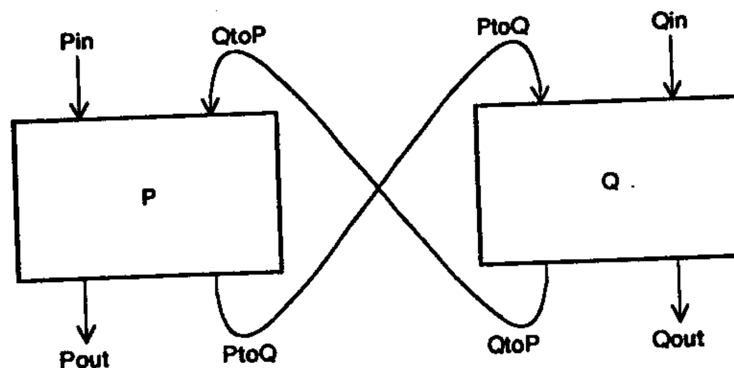
more general data structure than the streams of Landin, Kahn, or Weng.

The EditLine example illustrates a stream function with a "conventional" subroutine relationship to its caller. However, stream functions can also be used to specify co-routines or even networks of processes. Figure 2 gives a rather simple interconnection of two stream procedures, P and Q. There are many ways in which this interconnection could be implemented. Generally the implementation will be determined by the specific area of application. For example, P could be a program and Q, a file server, executing inside a small single-user computer. On the other hand, this interconnection could be adapted to our video terminal example. P could be a terminal and Q, the computer system to which it is attached. In addition, the interconnection need not represent two closely-coupled systems. It could represent a widely-dispersed network: P could be the account management system for a bank in London, and Q could be a similar system for a bank in Tokyo, and the arcs between the systems could be underwater cables or satellites. In Id this cyclic interconnection can be quite naturally specified as:

```
Pout, PtoQ ← P (Pin, QtoP) ;
Qout, QtoP ← Q (Qin, PtoQ)
```

The four stream operators previously described and Id's iterative *for-each* construct are all *determinate*, that is, for each possible set of input stream arguments there is only one possible set of output stream results. Determinacy has some very important consequences. First, any program written with determinate operators or any network composed from determinate modules is itself determinate. Second, all determinate modules are time-independent: inputs for a determinate module can be arbitrarily delayed without affecting that module's eventual results. These two properties of determinacy combine to reduce the difficulty of debugging stream programs, for they insure that any stream computation is repeatable. However, they also prevent the writing of either time-dependent or non-determinate computation, and thus it is reasonable to ask whether or not there are any inherently non-determinate applications of sufficient importance to justify the incorporation of non-determinate operators or constructs into a dataflow programming language. It should be noted that, except for machine languages in which indeterminacy results from interrupt processing, almost all conventional programming languages (including FORTRAN, Algol, Pascal, LISP, Cobol, and PL/I) are

Figure 2.



Interconnected network of P and Q

determinate. Concurrent Pascal and Ada are perhaps the only widely-known programming languages in which non-determinate computation can be expressed.

3. Non-determinate Computation

Resource allocators are part of many interesting application areas, such as transaction systems or operating systems. To see the necessity of non-determinate computation, let us consider a simple one-account, two-teller bank. The bank's accounting system processes streams of requests from and sends streams of replies to the two tellers. The tellers are competing for resources, in this case the dollars of the banks accounts, just as processes in a multi-user computer systems compete for resources such as tape drives. Suppose the tellers entered separate requests for the last dollar. We expect one teller to be granted the dollar, and the other to be denied it; however, we certainly do *not* expect, or want, that arbitration to be determinate. For if it were determinate, and consequently totally time-independent, the same teller would always "win" that particular race for the last dollar, even if the loser's request preceded his by hours or even by years. In reality the cost of determinacy is even higher: since the account management system is not prescient, it cannot know whether or not the preordained winner will ever wish to request the dollar, and thus the losing teller must always lose, even if the winner never enters the race. It is for these reasons, that determinate computation is inadequate for resource allocation problems.[1]

The smallest extension for obtaining non-determinism is the addition of a single non-deterministic stream operator, the *merge*. The *merge* receives two input streams and non-deterministically interleaves them into one output stream. The intended implementation of *merge* is an operator which waits for a value to appear on either of its input ports and then produces the received value at its output port and returns to its waiting state. Thus, there are three possible output stream responses when *merge* is applied to the input streams [5, 6] and [7]:

merge([5, 6], [7]) = [5, 6, 7] or [5, 7, 6] or [7, 5, 6]

Obviously, *merge* is not a function. In the presence of feedback, the choices of the *merge* may be further constrained, and Brock and Ackerman [5] have proven that the *merge* cannot even be formally represented as a relation on input and output streams. Given two terminal input streams TTY1 and TTY2, the *merge* operator and EditLine, our previously written stream program, can be used in the expression:

merge(EditLine(TTY1), EditLine(TTY2))

to interleave the commands (lines) typed on the two terminals into one. Within a operating system, this combined command stream could then be passed to stream-based resource allocators, often written with only determinate operators, and acted upon without the very serious problems encountered using totally determinate resource allocators.

---

1. Admittedly, there are imaginative schemes for restoring determinacy; but, all such schemes involve imposing somewhat troublesome (and ridiculous) prerequisites on system users. For example, the system could require that every user produce a (possibly null) output character every milli-second, and then it could poll its users in a round-robin fashion.

The two teller bank system can be written using *merge*'s to resemble the network of Figure 3. This network is composed of four types of modules, only one of which need be indeterminate. Each Tag–*i* module receives one input request stream, and tags each of its values with the number *i*. In Id this is simply written as:

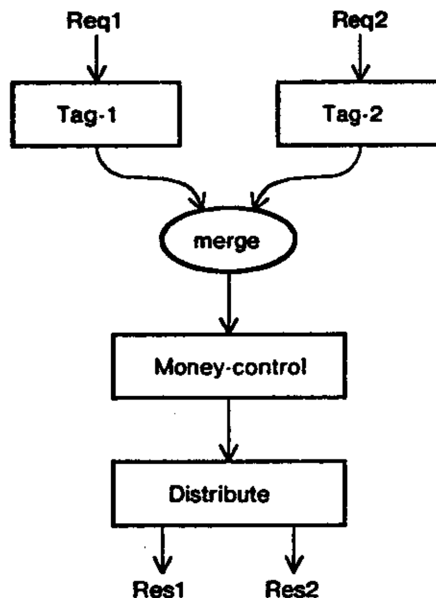
```
procedure Tag–i(X)
  ( for each x in X do
    return all <i, x> )
```

The tagged requests are then interleaved by a *merge* operator to form a single, combined stream of requests. This stream is next sent to the resource controller, Money–control, a stream procedure "containing" the resource. This procedure examines the tagged request stream and produces a stream of tagged responses. Suppose this trusting bank does nothing but maintain account balances, without even checking for account overdrafts. All requests to the bank are integers. Positive integers represent deposits. Negative integers represent withdrawals (negative deposits). Each transaction is confirmed by giving the new account balance. Then the resource allocator is:

```
procedure Money–control(REQ)
  ( initial account–balance ← 0
    for each req in REQ do
      new account–balance ← account–balance + req[2] ;
      confirmation ← new account–balance
    return all <req[1], confirmation> )
```

Distribute, the remaining module of the two teller bank system, is a determinate module which removes the tags and routes untagged responses to the appropriate recipients. With this "template" many resource allocation system may be programmed merely by replacing the resource controller module. The applicative

Figure 3.



Res1, Res2 ← Distribute(Money–control(*merge*(Tag–1(Req1),Tag–2(Req2))))

Two teller bank system

airline reservation system of Dennis [8] is constructed in this way.

While, in theory, the *merge* satisfies our requirements for non-determinism; in practice, it is clumsy to use because of the difficulties of connecting a resource allocator to its users. Interconnection with *merge*'s alone requires of each user an input-output stream pair for each resource it uses, and of each allocator an input-output stream pair for each user it services, and that requirement is a difficult, if not impossible, to satisfy in a flexible system. For while it is straightforward to extend this bank account system to service any fixed number N of tellers; it would be difficult, if not impossible, to extend it to arbitrary N *or* varying N. For this reason the language Id has managers [2, 3]: a high-level program construct, encompassing the previously presented allocator template, for specifying non-determinate computation. A related data construct has been proposed by Jayaraman and Keller [12] to solve the same problem.

From the viewpoint of its users, an Id manager resembles a normal (*i. e.* non-stream) function. The manager is "invoked" with single requests and yields single replies. This is very different from the resource allocator described in the previous example which was applied to streams of requests to yield streams of replies. In its calling sequence, the Id manager is quite conventional — resembling the calling sequence of Hoare's [11] monitors.

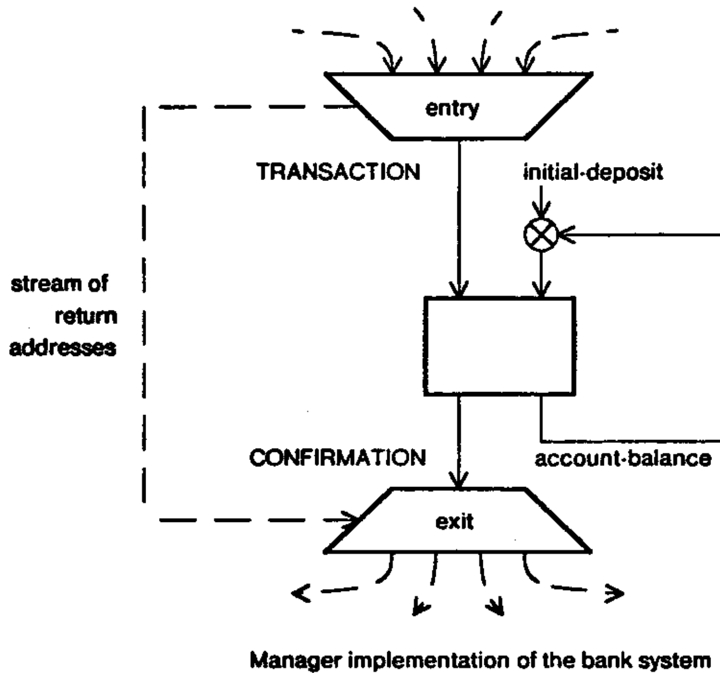
Linguistically, the manager is one or more Id assignments surrounded by an *entry-exit* pair. The assignment specifies a stream computation, the *entry* clause specifies how manager requests are formed into the input streams of the assignment, and the *exit* clause specifies how manager replies are formed from the output streams of the assignment. Below is the Id manager (illustrated in Figure 4) for the trusting bank account system.

```
bank ← manager(initial-deposit)
        ( entry TRANSACTION do
          CONFIRMATION ← ( initial account-balance ← initial-deposit
                           for each transaction in TRANSACTION do
                             new account-balance ← account-balance + transaction ;
                             confirmation ← new account-balance
                           return all confirmation )
          exit CONFIRMATION )
```

The *entry* clause of the bank manager causes all manager requests to be merged into a stream TRANSACTION, while the *exit* clause causes the values of the stream CONFIRMATION to be distributed as responses to the requests. To insure that each requester receives the proper response, it is very important that one response is produced for each request and that the *i*'th element of the response stream be, in fact, the intended response to the *i*'th request. Detailed implementations, at the dataflow machine level, for *exit-entry* have been described by Arvind, Gostelow, and Plouffe [3] and by Catto and Gurd [7].

In a computation there may be several instances of a manager. Each instance is created with, naturally enough, the *create* operator, so that *create*(M, V) makes an new instance of M with initial state V. Once created, a manager may be passed to many computations which then share it and thusly share the resource it protects. To send a request X to a manager M, the operation *use*(M, X) is performed. The operation returns the manager response.

Figure 4.



Manager implementation of the bank system

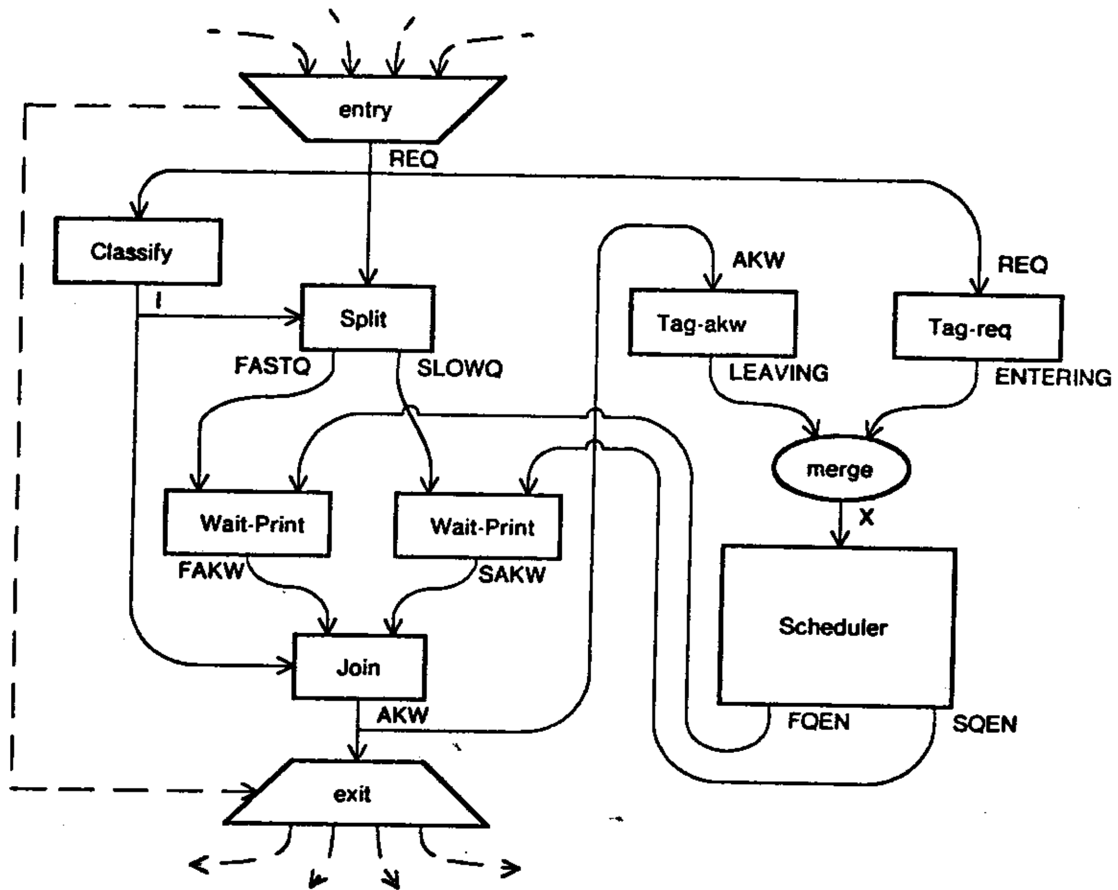
When the manager is used by many independent computations (or processes) the order in which requests arrive at the manager are unpredictable, and the *entry* clause serves as a non-deterministic *merge* of several "streams" of user requests. Often the indeterminism implicit within the *entry* will be the only indeterminism occurring within managers, although for sophisticated scheduling strategies use of the *merge* operator may be required within the manager body.

4. Scheduling with Managers

Unlike the simple bank system, many resource allocation problems require the scheduling of resource utilization according to request type, user priority, or resource state. In these situations, a resource scheduler is needed. The scheduler is a stream procedure which receives streams derived from two sources: the requests to the manager and the responses of the protected resource. The scheduler produces streams which enable resource access.

A manager using a scheduler is shown in Figure 5. This manager controls a printer. Messages of the form <size–of–file, file> are sent to the manager. Short print jobs, those less than ten pages, are rewarded absolute priority in printing. The stream procedure Classify examines the stream of manager requests and produces an output stream whose *i*'th element is 1 if the *i*'th request is short and 2 if the *i*'th request is long.

Figure 5.



Priority printer manager

```
procedure Classify(REQ)
  ( for each req in REQ do
      i ← (if req[1] < 10 then 1 else 2)
    return all i ) ;
```

The stream procedure Split uses the output of Classify to divide the input request stream into one stream of short requests and another of long requests.

```
procedure Split (I, X)
  ( for each i in I ; x in X do
      x1, x2 ← ( if i = 1 then x, λ else λ, x)
    return all x1 but λ, all x2 but λ )
```

The split request streams are examined by separate resource controllers which we will examine in more detail shortly. The responses of the controllers are joined by the procedure Join. The output of Split is used to insure that the responses are joined in the same manner in which they were split.

```
procedure Join (I, X1, X2)
  ( if first(I) = 1 then cons(first(X1), Join(rest(I), rest(X1), X2))
                    else cons(first(X2), Join(rest(I), X1, rest(X2))) )
```

It may be the case that an $i+1$'th request is short and processed before a long i'th request. An implementation of *cons* which is non-strict in *both* arguments is needed if the response for the short request is allowed to "leave" the manager before the response to the long request. This degree of non-strictness does not occur in the Weng implementation of streams in which individual stream values are produced strictly in order of their position within the stream, but can be obtained using the U-interpreter [3] implementation of streams where individual stream values are tagged with their position in the stream and then produced in an arbitrary order. The U-interpreter also allows stream producing iterations to be overlapped. This same degree of non-strictness is also available when streams are implemented using the demand-driven interpreters for the functional languages LISP[9, 10, 14], Lucid [4], and SASL [17].

Actual resource allocation occurs within the Wait-*f* procedures. These procedures, one for each request class, simply delay the application of a procedure *f* until an enabling value is available on an associated enabling stream.

```
procedure Wait-f(Q, QEN)
  ( for each req in Q ; enable in QEN do
      result ← f(req)
    return all result )
```

The protected procedure *f* may itself be a "call to" (*i. e.*, *use* of) another manager, and in this manner managers may be recursive and arbitrarily nested.

The enabling streams are produced by Scheduler which generates them using as its input stream the merged tagged streams of requests to the manager and responses from the protected resource. Momentarily assuming the existence of the scheduler, the manager of Figure 5 can be programming in Id as:

```
manager Printer
  ( entry REQ do
      I ← Classify(REQ) ;
      FASTQ, SLOWQ ← Split(I, REQ) ;
      FAKW ← Wait-print(FASTQ, FQEN) ;
      SAKW ← Wait-print(SLOWQ, SQEN) ;
      AKW ← Join(I, FAKW, SAKW) ;
      ENTERING ← Tag-req(REQ) ;
      LEAVING ← Tag-akw(AKW) ;
      X ← merge(ENTERING, LEAVING) ;
      FQEN, SQEN ← Scheduler(X)
    exit AKW )
```

The guts of the manager lie in its scheduler: the remainder is simply a highly structured interconnection of trivial stream procedures. Obviously, more syntax could be developed to aid the manager programmer.

The Scheduler receives a stream of tagged values. Inputs of the form <"req", <file-size, file>> are requests to the manager. Other inputs are acknowledgments from the controlled printer resource. The first field of other inputs is "akw". The scheduler is generating two output enabling streams (FQEN and SQEN),

which are routed to the fast and slow queues of the printer. The task of the scheduler is to make sure that fast jobs are printed before slow ones. It does this by watching the printer acknowledge stream to determine when previously enabled requests have been serviced. The scheduler needs a state of three internal variables to accomplish its task:

- **busy** — is true if the printer is servicing a request,
- **nfq** — is the number of queued (waiting) fast jobs, and
- **nsq** — is the number of queued slow jobs.

The Scheduler itself is nothing but a seven case analysis of the state of the printer and of the requests.

```
FQEN, SQEN ← procedure Scheduler(X)
  ( initial nfq, nsq ← 0, 0 ;
        busy ← false

   for each x in X do
     fqen, sqen, new nfq, new nsq, new busy ←
       ( if x[1] = "akw" then
           ( if nfq = 0 ∧ nsq = 0 then λ, λ, 0, 0, false
             elseif nfq = 0 ∧ nsq > 0 then λ, "go", 0, nsq−1, true
             !!! nfq > 0 !!! else "go", λ, nfq−1, nsq, true )
        !!! x is a request !!! else
           ( if busy then ( if x[2,1] < 10 then λ, λ, nfq + 1, nsq, true
                                           else λ, λ, nfq, nsq + 1, true )
                  else ( if x[2,1] < 10 then "go", λ, 0, 0, true
                                           else λ, "go", 0, 0, true )))

   return all fqen but λ, all sqen but λ )
```

Let us consider one of the cases. Suppose the fast queue is empty (nfq = 0), the slow queue is not (nsq ≠ 0), and the next tagged input is a printer acknowledgment. In this situation, the second case of Scheduler, a queued slow request is enabled (with the string "go") and the slow queue counter is decremented. The other cases are similar.

Obviously, the scheduler is the most difficult part of writing the printer manager. It is here that possibility of semantic error is greatest. For example, suppose we had initialized the variable busy to *true* (and originally we had). In that case the manager would appear to deadlock. It would always queue its requests without ever sending them to the printer and, consequently, without ever receiving any acknowledgments from the printer. The construction of good invariants can significantly reduce the number of programming errors. Our invariants for the scheduler, written using the internal variables busy, nfq, and nsq of Scheduler and stream variables FASTQ, SLOWQ, FQEN, SQEN, and X of the Printer manager, are:

```
number-of-fast-requests-in(X) = size-of(FASTQ) = size-of(FQEN) + nfq
number-of-slow-requests-in(X) = size-of(SLOWQ) = size-of(SQEN) + nsq
number-of-acknowledgments-in(X) = size-of(FQEN) + size-of(SQEN), if ~busy
                                = size-of(FQEN) + size-of(SQEN) − 1, if busy
```

Id managers are not a syntactic solution to all the pitfalls seen in many resource allocation systems. In particular, it is possible to write "bad" managers which deadlock. However, Wadge [18] has developed criterion for determining the existence of potential deadlock which are applicable to a wide class of determinate systems.

5. Conclusions and Scope for Further Research

Thus far we have presented the reader with a discussion of the problems that the incorporation of history sensitive and non-determinate computation into applicative languages posed, and have advanced streams and managers as solutions to those problems. However, we must include one caveat (or perhaps one challenge for the ambitious reader): Managers are not completely functional and can introduce some of the same problems found in object-oriented programming. For example, because a manager may be passed a reference to itself, it is possible to form a cycle in the manager's machine-level representation and to, thus, complicate reference count garbage collection. Managers even allow us to simulate our old nemesis, the memory cell: In particular, the following manager is a memory cell receiving read requests of the form <"read", V> and write requests of the form <"write", V>:

```
cell ← manager (InitV)
      ( entry REQ do

        RES ← ( initial V ← InitV
                for each req in REQ do
                  new V, res ←
                    ( if req[1] = "read" then
                        V, V
                      ||| req[1] = "write" ||| else
                        req[2], V )
                return all res) )

        exit RES )
```

"We have met the enemy, and he is us!"
Pogo, Walt Kelley

With this manager, memory cell address may be created merely by the call *create*(cell, initial-V).

Research must be directed toward the development of methodologies for reasoning about the correctness of managers or, for that matter, any of the "less applicative" constructs for specifying inter-process communication and sharing of resources. We believe our emphasis on streams will enable us to develop more successful strategies than are possible with totally object-oriented constructs. For example, we can write invariants on the *history* of communication by *direct* use of the variables of our program as opposed to using unguaranteed encodings of program history into non-stream variables. We also believe that streams will prove to be a more natural basis for developing formal semantics for that very difficult domain of indeterminate communication. Already, Brock and Ackerman [5] have developed *scenarios* and Kosinski [15] has developed *tagged sequences* to represent dataflow streams. Such formalisms should, in turn, prove useful in developing a bases for formal reasoning about the correctness of programs using streams and managers.

6. References

[1] Ackerman, W. B., "Data Flow Languages", *Computer 15*, 2(February 1982), 15–25.

[2] Arvind, K. P. Gostelow, and W. Plouffe, "Indeterminacy, Monitors and Dataflow", *Proceedings of the Sixth ACM Symposium on Operating Systems Principles, Operating Systems Review 11*, 5(November 1977), 159–169.

[3] Arvind, K. P. Gostelow, and W. Plouffe, *An Asynchronous Programming Language and Computing Machine*, Department of Information and Computer Science (TR 114a), University of California - Irvine, Irvine, California, September 1978.

[4] Ashcroft, E. A., and W. W. Wadge, "Lucid, a Nonprocedural Language with Iteration", *Communications of the ACM 20*, 7(July 1977), 519–526.

[5] Brock, J. D., and W. B. Ackerman, "Scenarios: A Model of Non-determinate Computation", *International Colloquium on Formalization of Programming Concepts, Lecture Notes in Computer Science 107*, April 1981, 252–259.

[6] Burge, W. H., *Recursive Programming Techniques*, Addison-Wesley Publishing Co., Reading, Massachusetts, 1975.

[7] Catto, A. J., and J. R. Gurd, "Resource Management in Dataflow", *Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture*, October 1981, 77–84.

[8] Dennis, J. B., "A Language Design for Structured Concurrency", *Design and Implementation of Programming Languages: Proceedings of a DoD Sponsored Workshop, Lecture Notes in Computer Science 54*, October 1976, 231-242.

[9] Friedman, D. P., and D. S. Wise, "CONS Should Not Evaluate its Arguments", *Automata, Languages, and Programming: Third International Colloquim* (S. Michaelson and R. Milner, Eds.), July 1976, 257–284.

[10] Henderson, P., *Functional Programming: Application and Implementation*, Prentice/Hall International, Englewood Cliffs, New Jersey, 1980.

[11] Hoare, C. A. R., "Monitors: An Operating System Structuring Concept", *Communications of the ACM 17*, 10(October 1975), 549–557.

[12] Jayaraman, B., and R. M. Keller, "Resource Control in a Demand-driven Data-flow Model", *Proceedings of the 1980 International Conference on Parallel Processing*, August 1980, 118–127.

[13] Kahn, G., "The Semantics of a Simple Language for Parallel Programming", *Information Processing 74: Proceedings of IFIP Congress 74*, August 1974, 471–475.

[14] Keller, R. M., G. Lindstrom, and S. S. Patil, "A Loosely-Coupled Applicative Multi-processing System", *Proceedings of the 1979 National Computer Conference, AFIPS Conference Proceedings 48*, June 1979, 613–622.

[15] Kosinski, P. R., "A Straightforward Denotational Semantics for Non-Determinate Data Flow Programs", *Conference Record of the Fifth ACM Symposium on Principles of Programming Languages*, January 1978, 214–221.

[16] Landin, P. J., "A Correspondence between ALGOL 60 and Church's Lambda Notation: Part I", *Communications of the ACM 8*, 2(February 1965), 89–101.

[17] Turner, D. A., "A New Implementation Technique for Applicative Languages", *Software - Practice and Experience 9*, 1(January 1979), 31–49.

[18] Wadge, W. W., "An Extensional Treatment of Dataflow Deadlock", *Theoretical Computer Science 13*, 1(January 1981), 3–15.

[19] Weng, K.-S., *Stream-Oriented Computation in Recursive Data Flow Schemas*, Laboratory for Computer Science (TM-68), MIT, Cambridge, Massachusetts, October 1975.