PURELY FUNCTIONAL OPERATING SYSTEMS

Peter Henderson

University of Newcastle upon Tyne

ABSTRACT

A range of operating systems are implemented, the entire text of each system being written in a purely functional style. These implementations lend themselves to configuration on a distributed collection of computers. Each computer is assumed to support a reasonable implementation of a purely functional language.

1. KEYBOARDS AND SCREENS

Consider a program which will accept an infinite sequence of integers typed at a keyboard and will display on a screen twice the value of each integer soon after the integer has been completely typed. The integers which appear on the screen also constitute an infinite sequence. Such a program can be specified in a purely functional style as follows:

```
screen = double(keyboard)
         where double(x) = cons(2Xhd(x),double(tl(x)))
```

A reasonable implementation of a purely functional language might be expected to execute this program as follows. The elements of the sequence "screen" are displayed as soon as may be possible. Thus the first element is displayed as soon as sufficient keystokes have been made to guarantee that the integer which is the first element of the sequence "keyboard" has been completed, say as soon as a blank has been typed.

Suppose now we wish to attach two keyboards to this program, in such a way that the display on the screen is an interleaving of the sequences obtained by doubling each of the values typed at each of the keyboards. Then we might have the following purely functional program:

```
screen = double(interleave(keyboard1,keyboard2))
         where double(x) = cons(2Xhd(x),double(tl(x)))
```

Again, we would require of any reasonable implementation that the elements of the sequence "screen" might appear as soon as possible after the corresponding integers have been typed at the relevant keyboard. This means that, soon after an integer has been completed at either keyboard, the corresponding value should appear on the screen. Since the program can have no control over the speed at which integers are typed at each keyboard, indeed one or other may even be idle for a while, the function "interleave" should be defined so that its result interleaves the elements of its two argument sequences in the order in which they first become available. Thus "interleave" is not a function only of the elements of the sequences which constitute its arguments but also of the time at which each element becomes completely defined. It is my purpose in this document to show how a function such as "interleave" can be deployed in programming a range of useful systems, and then to discuss whether or not it should be included in an otherwise purely functional language. The programs which we shall write here retain most of the pleasant properties of purely functional programs.

With this preparation, let us be more precise about the definitions of the functions we have so far introduced. We shall make use of the very powerful notation of S-expressions for data in our examples. The functions hd and tl will be used in preference to car and cdr as the basic operations upon S-expressions. When a sequence of data items are typed at a keyboard, each item will in general be an S-expression and the entire sequence itself will be represented by a semi-infinite list. Consider the simple program

```
screen = interleave(keyboard1,keyboard2)
```

and suppose that the items typed at each keyboard are as follows:

```
keyboard1 = (127 (PUT FRED (BLAH BLAH)) (GET F) 18 19 ...

keyboard2 = (-14 (RUN F G H) (SAVE F) (SAVE G) (1 2 3 4) ...
```

A possible sequence in which these items might appear on the screen is:

```
screen = (127 -14 (RUN F G H) (PUT FRED (BLAH BLAH))
          (SAVE F) (GET F) (SAVE G) 18 19 (1 2 3 4) ...
```

Note first of all that each of the items in the sequences has remained intact and secondly that, while the sequences as typed on the keyboards have been interleaved, their order has been retained. That is to say, each of the keyboard sequences can be obtained by deleting certain of the items in the screen sequence.

If T is the type of a data item (e.g. integer, symbolic atom, list, etc.) then let seq(T) denote the type of a (possibly semi-infinite) sequence of items, each of type T. In the previous examples, our functions had the following types:

```
hd         :  seq(T) → T
tl         :  seq(T) → seq(T)
cons       :  T × seq(T) → seq(T)
double     :  seq(integer) → seq(integer)
interleave:  seq(S-expression) × seq(S-expression) →
                    seq(S-expression)
```

Now, let us give a more precise definition of the "interleave" function. If x and y are both of type seq(T) then interleave(x,y) is of type seq(T). Further, one or other of the following situations pertains:

i) hd(interleave(x,y)) = hd(x)
   tl(interleave(x,y)) = interleave(tl(x),y)

ii) hd(interleave(x,y)) = hd(y)
    tl(interleave(x,y)) = interleave(x,tl(y))

Thus we see that any items appearing in interleave(x,y) come from either x or y and that further the order of items in x and y is preserved in interleave(x,y). In the final section we give a possible implementation of "interleave". Suffice it to say here that "interleave" will be implemented in such a way that, in the systems which follow, each sequence will be consumed at the rate at which it is generated.

2. SIMPLE DATABASES

We shall define a very simple database system which allows its user to save and recall "files". We shall assume a file is an S-expression and a filename is a symbolic atom. The database will be represented by a linear list of (filename file) pairs, for simplicity. In practice a tree structure would be more acceptable. We introduce the following functions as basic:

```
put:  filename × file × database → database

get:  filename × database → file
```

with the following definitions

```
put(f,s,db) ≡ if db=NIL then cons(cons(f,s),NIL) else
   if hd(hd(db)) = f then cons(cons(f,s),tl(db)) else
                          cons(hd(db),put(f,s,tl(db)))

get(f,db) ≡ if db=NIL then MISSING else
   if hd(hd(db)) = f then tl(hd(db)) else get(f,tl(db))
```

We see that put(f,s,db) simply updates the list db to incorporate the pair (f s), eliminating any other pair with the filename f, while get(f,s,db) returns the s corresponding to f in db.

Now assume the user of our database is allowed to present two commands. The first has the form (PUT f s) where f is a filename and s a file, and has the effect of adding the file s to the database, with name f. The second has the form (GET f) and has the effect of retrieving the file with filename f. This behaviour can be encapsulated in a single function.

dbstep: command × database → response × database

which denotes the application of a single command to the database. We define

```
dbstep((PUT f s),db) = DONE,put(f,s,db)
dpstep((GET f),db) = get(f,db),db
```

Note that the response to a PUT command is the atom DONE, while the response to a GET command is the file (or the atom MISSING if no such file is found, see "get").

We can implement our database system as a single function by allowing it to map a sequence of commands to a sequence of responses. Let us define

dbf: seq(command) → seq(response)

as follows:

```
dbf(c) ≡ dbf1(c,NIL)
dbf1(c,db) ≡ cons(m,dbf1(tl(c),db'))
                  where m,db' = dbstep(hd(c),db)
```

This is a fairly conventional recursive definition using a subsidiary function dbf1. For a given database, db, the function dbf1 computes, for the first command, the first response m and the new database db'. The sequence of responses from dbf1 is then m, followed by responses invoked by the remaining commands when presented to db'.

We can immediately see how dbf could be deployed as a single user database system

```
screen = dbf(keyboard)
```

Here the sequence of commands typed at the keyboard invoke a sequence of responses on the screen. If we display the sequence interleave(keyboard,screen) then this might take the following value:

```
(PUT FRED (BLAH BLAH))
DONE
(GET FRED)
(BLAH BLAH)
(GET MAVIS)
MISSING
(PUT MAVIS (1 2 3 4))
DONE
(GET MAVIS)
(1 2 3 4)
(GET FRED)
(BLAH BLAH)
     .
     .
     .
```

We assume the interaction with dbf goes on for ever, despite the likely switching on and off of equipment.

In order to allow two users to share the database, we must interleave their keyboards, for example:

```
screen = dbf(interleave(keyboard1,keyboard2))
```

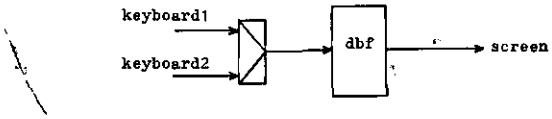
This means that the two users, having separate keyboards, must share a single screen. We shall give this program a name, in order that we may refer to it later. It is the first in a series of useful systems, so we shall call it sys0. We have

```
sys0: seq(command) × seq(command) → seq(response)
```

where we define:

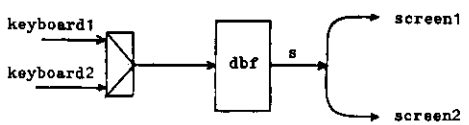
```
sys0(keyboard1,keyboard2) = screen
   where screen = dbf(interleave(keyboard1,keyboard2))
```

Also, for pedagogical purposes we shall introduce a diagram for each system. In the diagrams, arrows will always represent sequences and boxes will always represent functions. Thus the definition of each system can be read, in a straightforward manner from the diagram. In what follows we shall give each system in both forms. The diagram for sys0 is as follows:



The interleave function is so common that we have given it a special box, with a mnemonic triangle in it. Each of the arrows in the diagram denotes a sequence of values. Each of the boxes is a function from its incoming sequences to its outgoing sequences.

If we wished to give each user his own personal copy of the screen, we would define the following systems, where we assume keyboard1 is associated with screen1 and keyboard2 with screen2:



Reading the definition from this diagram, we construct

```
sys1(keyboard1,keyboard2) = screen1,screen2
   where screen1 = s
     and screen2 = s
     and s = dbf(interleave(keyboard1,keyboard2))
```

Whilst these systems do represent true sharing of a database, the latter version is a little unrealistic in that each screen reflects the responses to activities on the other keyboard.

To solve this problem we introduce the notion of tagging the commands to and responses from the database. Let us use the functions:

```
tag(t,x) ≡ cons(cons(t,hd(x)),tag(t,tl(x)))

untag(t,x) ≡ if hd(hd(x))=t then cons(tl(hd(x)) ,untag(t,tl(x)))
                           else untag(t,tl(x))
```

We have that, if x is a sequence of items then tag(t,x) is the same sequence, each item having been tagged with the atom t. The function untag(t,x) is used to project the sequence of tagged items x onto the sequence of untagged items whose tags in x are equal to t. We use the special boxes shown below to denote these functions in our diagrams.

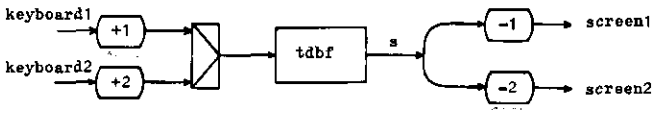
We must modify the database function to preserve tags, which is accomplished very simply by altering dbstep as follows:

```
tdbstep((t PUT f s),db) = (t DONE),put(f,s,db)
tdbstep((t GET f),db) = (t get(f,db)),db
```

We redefine dbf in terms of this new step function, to derive a tag preserving database.

```
tdbf(c) ≡ tdbf1(c,NIL)
tdbf1(c,db) ≡ cons(m,tdbf1(tl(c),db'))
                  where m,db' = tdbstep(hd(c),db)
```

In terms of this we can define a system which allows two users to properly share the database without seeing each others responses.

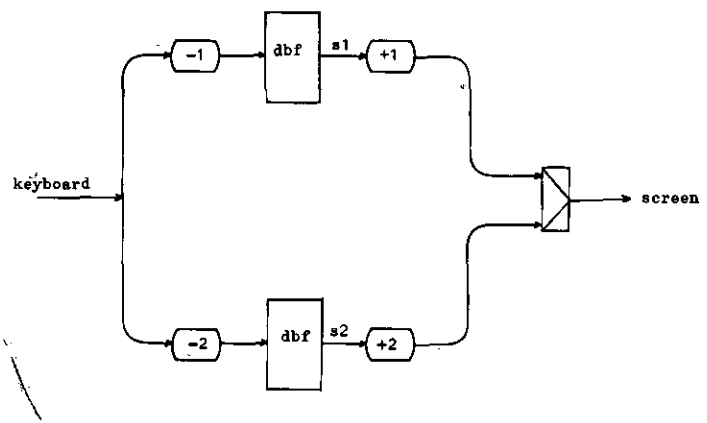


The system can be written out more formally as:

```
sys2(keyboard1,keyboard2) = screen1,screen2
     where screen1 = untag(1,s)
       and screen2 = untag(2,s)
       and s = tdbf(interleave(tag(1,keyboard1),
                               tag(2,keyboard2)))
```

Now, with this system each user of the database sees only the responses associated with his requests.

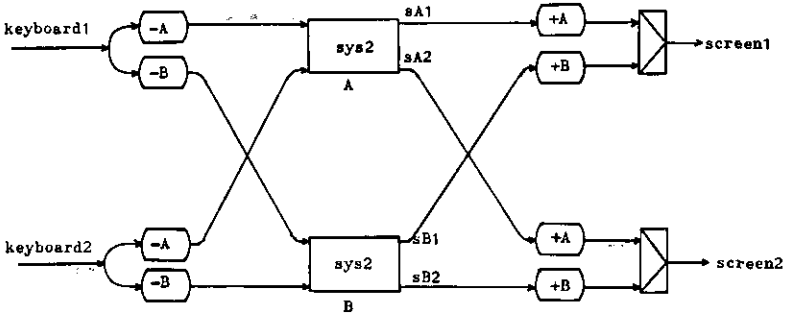
Already we have sufficient primitives to begin to build a range of different and interesting systems. Let me illustrate just two possibilities. The following system gives simultaneous access from a single keyboard to two independent databases.



```
sys3(keyboard) ≡ screen
    where screen = interleave(tag(1,s1),tag(2,s2))
      and      s1 = dbf(untag(1,keyboard))
      and      s2 = dbf(untag(2,keyboard))
```

Note how the user must tag all his commands at the keyboard with the identity of the database he wishes to access, and that responses come back with an appropriate tag attached. Note also that commands with tags other than 1 or 2 are simply ignored.

We can combine the structures of sys2 and sys3 in an obvious way to construct a database system which allows two users to share access to two independent databases.




```
sys4(keyboard1,keyboard2) ≡ screen1,screen2
    where screen1 = interleave(tag(A,sA1),tag(B,sB1))
      and screen2 = interleave(tag(A,sA2),tag(B,sB2))
      and sA1,sA2 = sys2(untag(A,keyboard1),untag(A,keyboard2))
      and sB1,sB2 = sys2(untag(B,keyboard1),untag(B,keyboard2))
```

This system allows the users to identify the database which they wish to access with the tags A and B. Inside sys2, tags 1 and 2 are used to identify which user is intended to receive the response, these tags are removed before the response streams sA1, etc. are generated. The responses are properly tagged with the name of the database from which they come.

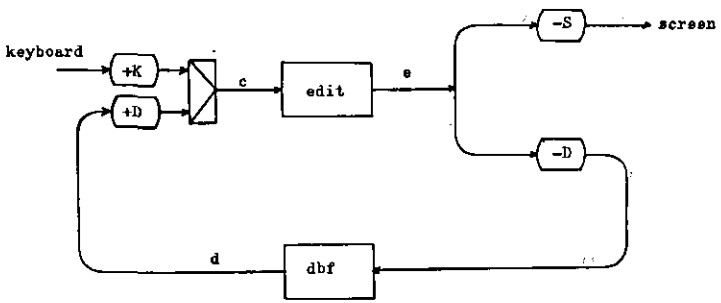
Note that sys4 has precisely the same structure as sys2 and that a yet more elaborate system, allowing two users to access up to four independent databases could be generated simply by replacing the occurrences of sys2 in the above diagram by sys4.

3. EDITORS AND DATA EXCHANGES

We now wish to address the problem of getting information out of a database, altering it and putting it back in again. We will implement only the most trivial of editors, which will have only one command for changing the structure of a file. A more useful editor would require a full range of commands, allowing an S-expression to be explored and altered at will. This is not an issue here. The editor serves as a data exchange in the sense that files wait there before being sent to the database and are held there also on being retrieved from the database. We think of the editor as containing therefore a "register" x, in which at any time a file is held. The commands a user may give are:

(PUT f)	which puts x in the database with name f
(GET f)	which replaces x by the contents of the file named f
(CHANGE t1 t2)	which changes x according to templates t1 and t2
PRINT	which displays the current contents of x on the screen

We intend our edit function to fit into the following system:




```
sys5(keyboard) ≡ screen
    where screen = untag(S,e)
      and e = edit(c)
      and c = interleave(tag(K,keyboard),tag(D,d))
      and d = dbf(untag(D,e))
```

Here we have used interleave with appropriate tags to merge the input from the keyboard and from the database to the editor. The edit function will therefore see tagged commands of the form:

```
(K GET f)
(K PUT f)
(K CHANGE t1 t2)
(K.PRINT)
(D.s)          where s is some file (S-expression)
```

and will produce tagged responses of the form

```
(S.m)          where m is a message
(D PUT f s)
(D GET f)
```

The edit function takes on much the same structure as dbf, in that we consider it applying a sequence of commands (to x) one step at a time.

```
edit(c)    ≡ edit1(c,NIL)
edit1(c,x) ≡ cons(m,edit1(t1(c),x'))
                 where m,x' = editstep(hd(c),x)
```

We have introduced functions of the following types.

```
    edit: seq(tagged-command) → seq(tagged-response)
   edit1: seq(tagged-command) × file → seq(tagged-response)
editstep: tagged-command × file → tagged-response × file
```

Now we are in a position to define editstep by enumerating the cases which it will encounter.

```
editstep((K GET f),x) = (D GET f),x
editstep((K PUT f),x) = (D PUT f x),x
editstep((K CHANGE t1 t2),x) = (S dump(x')),x'
                                 where x' = change(x,t1,t2)
editstep((K.PRINT),x) = (S.x),x
editstep((D.DONE),x) = (S DONE),x
editstep((D.s),x) = (S NEWFILE),s
```

This is a very trivial editor. We have not defined the functions

```
change: file × template × template → file
dump:   file → response
```

which we leave to the imagination of the reader. In practice an editor would need a number of commands and here CHANGE is being used to illustrate how they might be implemented. Note how the editor forwards the GET command to the database and subsequently receives the new file from the database. There will be a period

of time, while the editor is waiting for this file, when commands applied to x would be accepted. It would be foolhardy to take advantage of this fact however, because the new file could arrive at any time, and obliterate x.

A possible session with this editor might be as follows:

```
(GET FRED)
(NEWFILE)
PRINT
(BLAH BLAH)
(CHANGE (1 2) (1 B B 2))
(BLAH B B BLAH)
(CHANGE 1 (1 1 1))
(* * *)
PRINT
((BLAH B B BLAH) (BLAH B B BLAH) (BLAH B B BLAH))
(PUT FRED)
DONE
```

Alternate lines are reflections of commands typed at the keyboard and responses displayed on the screen. We have taken a particular decision as regards the form of templates and the nature of a "dumped" expression which happens to reflect the behaviour of an editor with which we have some experience.

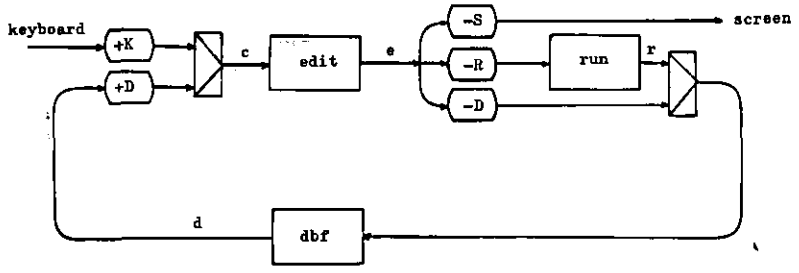
Finally, suppose we are able to place in the database the text of programs. That is to say, some of our files represent Lisp programs. Then, we might extend sys5 to include the capability for executing these programs as follows. We define functions to execute a sequence of runcommands using a conventional eval function.

```
    run: seq(runcommand) → seq(putcommand)
runstep: runcommand → putcommand
   eval: file → file

    run(c) ≡ cons(runstep(hd(c)),run(tl(c)))

    runstep((RUN s f)) = (PUT f eval(s))
```

It is necessary to add one more step to the editor.

```
editstep((K RUN f),x) = (R RUN x f),x
```

which will send to the run function the command to execute the program in x and place the result into the file called f. The system structure is then as follows:




```
sys6(keyboard) = screen
   where screen = untag(S,e)
     and e = edit(c)
     and c = interleave(tag(K,keyboard),tag(D,d))
     and d = dbf(interleave(r,untag(D,e)))
     and r = run(untag(R,e))
```

With this system, the user can GET programs from the database, CHANGE them (to include some data), send them to be RUN and inspect the results of execution by getting the appropriate file from the database. A session might be

```
(GET FACTORIAL)
(NEWFILE)
(CHANGE 1 (1 (QUOTE 6)))
(* *)
(RUN RESULTS)
... wait a respectable amount of time ...
(GET RESULTS)
(NEWFILE)
PRINT
72
```

As defined, the user can only guess when "run" will have completed its work, but there is nothing to prevent him from getting on with other work in the meantime. Requests to "run" will be queued by this system.

In fact, with a slightly more elaborate editor, this is quite a usable system, allowing as it does the construction and execution of programs and the collection of results in files. It is remarkable that virtually its entire definition is contained in this document in an executable form. Altogether we might expect each of the basic subsystems (i) edit, (ii) run, (iii) dbf, (iv) sys to be about 60 lines of purely functional program. For that, less

than 250 lines of formal text, we have an interactive operating system of some considerable power. Since this is only a beginning, and since pure functions are such a powerful programming medium, we might expect whole order of magnitude improvements in the power for marginal increases in the size of such systems.

4. HOW MANY PROCESSORS?

The style of definition of systems which we have used here has an important property. We can easily configure the system to run on a set of processors linked by some kind of serial communication lines. Consider sys6 for example. Clearly we could implement it on a single processor with serial lines to keyboard and screen. Alternatively, we could devote a separate computer to the function dbf. Since dbf is likely to require a great deal of storage space for its data structures, it is reasonable to assume that this storage will extend over secondary media and hence it is sensible to devote a separate processor to this task. Similarly, a separate processor could be devoted to the run task. It is an exciting prospect that purely functional systems of this kind could be configured in an arbitrary way with the greatest of ease.

5. THE INTERLEAVE "FUNCTION"

Finally, let us turn to the implementation of interleave, out of which all these systems have been built. In practice, we would expect interleave to behave in a demand driven fashion. That is to say, because of demand for its result, it constantly demands its arguments. This way of looking at a function such as interleave is intuitive and hence valuable. It is however an operational view and may lead us to attribute to interleave properties which we do not wish it to have. Rather, if we wish interleave to be a function and enjoy all the usual substitutivity properties of a function, then we must be very careful about its implementation.

Consider the following functions (cf. sys2)

```
p1(x,y) ≡ h(s1,s2)
          where  s1 = f(s)
            and  s2 = g(s)
            and   s = interleave(x,y)

p2(x,y) ≡ h(s1,s2)
          where  s1 = f(interleave(x,y))
            and  s2 = g(interleave(x,y))
```

The function p2 is derived from p1 by substituting for s. If interleave is a function, p1(x,y) = p2(x,y). That is, when

considered as functional programs p1 and p2 should produce exactly the same output when given the same input. In p2 the two occurrences of the subexpression "interleave(x,y)" must then denote the same sequence. There is a tendancy when thinking in terms of functions demanding their arguments to believe that the two functions p1 and p2, when considered as functional programs, will behave differently because of the extra time, no matter how small, required to evaluate interleave(x,y) twice. However, if interleave is a function, the only difference between p1 and p2 as programs will be that the elements of the sequence p1(x,y) = p2(x,y) may appear at different rates in each program. They will not appear in different order.

We could of course insist that interleave is a function. We must then answer two questions. Firstly, can the user of interleave write all the systems which he desires? Secondly, is the need to make interleave a function too much of a constraint on the implementor? Let us consider informally how interleave could be implemented as a function. One obvious approach is to arrange that all the items in the sequences which are arguments to or results of the interleave function are timestamped. The timestamps on successive items in a sequence would be non-decreasing, and interleave would maintain this invariant on its result sequence, as well as implementing the rest of its semantics. In case two items are encountered on the two argument sequences with identical timestamps, interleave can consistently give preference to its first argument (say). We would contrive to ensure that the other functions ignored the timestamps, but otherwise maintain the invariant on their result sequences. Finally, it is necessary, and even appropriate, if all the functions, including interleave, slightly delay items by incrementing the timestamps suitably, but we shall not pursue that detail here.

Certainly, all the programs presented in this paper behave appropriately with this definition of interleave. However, they are but a small class and not representative of the whole range of operating systems. The difficult question to answer is whether the explicit implementation of a timestamp mechanism is a reasonable proposition. It probably is not, although it deserves some experimentation . Alas, the implicit implementation of a timestamp mechanism, where the items are not actually stamped but take as their timestamp the time of inspection, does not implement interleave as a function.

If we implement interleave in a demand driven way, as we had intended all along, then our programs have a purely functional style but are no longer functions. This makes them more difficult to reason about and probably more difficult to make correct. Returning to the programs p1 and p2, we need only consider the case when f consumes its argument at a much greater rate than g. In p1, f will force s to become elaborated and g will of course

see the same sequence. In p2 however, there is no guarantee that, simply because f has forced one occurrence of interleave(x,y) to become elaborated, and presumably elaborated parts of x and y, that the second occurrence of interleave(x,y) will return the same (prefix of a) sequence. The problem is that in the argument position of f, interleave(x,y) produces a result dependent on the availability of elements in x and y while later (presumably) in the argument position of g, interleave will find these sequences already elaborated and will thus produce a result which may be determined by its bias to one argument.

The kinds of systems which we have demonstrated in this paper justify further study of the interleave primitive. It is very powerful, its only shortcoming being its non-determinism, but then, since we are trying to model non-deterministic behaviour it may be that a non-deterministic primitive is necessary.

ACKNOWLEDGEMENT

This work has benefited greatly from the comments of colleaques at the Universities of Newcastle and Edinburgh and from the members of IFIP WG2.3. In particular, Simon Jones has helped me to straighten out some of my thinking. This is not to suggest that any of those who were critical of my earlier presentations would not be equally critical of this revised one. My understanding in this area is far from adequate and I would very much appreciate further comment.

REFERENCES

Backus, J. (1978). Can Programming be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs Comm. A.C.M. 21, 8, pp.613-641.

Friedman, D.P., Wise, D.S. (1980). An Indeterminate Constructor for Applicative Programming. In Conf. Record of 7th Annual A.C.M. Symposium on Principles of Programming Languages, Las Vegas.

Friedman, D.P., Wise, D.S. (1977). Applicative Multiprogramming. Technical Report No. 72, Indiana University, Bloomington.

Kahn, G., McQueen, D. (1977). Coroutines and Networks of Parallel Processes. In Information Processing 77, North Holland, Amsterdam.

Keller, R.M., Lindstrom, G., Patil, S.S. (1979). A Loosely-Coupled Applicative Multiprocessing System. In AFIPS Proc. 48, pp. 613-621.

Keller, R.M., Lindstrom, G., Patil, S.S. (1980). Dataflow Concepts for Hardware Design. In COMPCON 80 (VLSI - New Architectural Horizons), IEEE Computer Society.