Technical Report

UCAM-CL-TR-56
ISSN 1476-2986

Number 56

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

A new scheme for writing functional operating systems

William Stoye

September 1984

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

Technical reports published by the University of Cambridge
Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

A New Scheme for Writing Functional Operating Systems.

William Stoye

University of Cambridge Computer Laboratory

Abstract

A scheme is described for writing nondeterministic programs in a functional language. The scheme is based on message passing between a number of expressions being evaluated in parallel. I suggest that it represents a significant improvement over previous methods employing a nondeterministic merge primitive, and overcomes numerous drawbacks in that approach. The scheme has been designed in a practical context, and is being used to write an operating system for SKIM, a functionally programmed machine. It is not yet well understood in a mathematical sense.

Keywords: functional programming,
functional languages,
functional operating systems.

Nondeterminism

Expressions in a functional language denote a value, and the evaluation of such an expression results in this value and has no other effect. If a function `f` is called with the same argument `x` in two different places in a functional program, the two calls will yield the same value.

One aspect of pure functional languages is their inability to behave nondeterministically. There are a number of reasons why nondeterministic behaviour might be desirable.

1. In order to write operating systems

The behaviour of an operating system seems to be nondeterminisitc in certain circumstances. These usually take the form of multiple simultaneous demands for attention, e.g. break keys, the control of background tasks and so on.

2. In order to be sufficiently abstract

There are circumstances when one of a collection of results will do, and having to make an arbitrary choice seems undesirable.

3. In order to represent certain objects correctly

The correct handling of infinite sets requires the ability to select a member of a set nondeterministically. It is possible that cases 1 and 2 are covered by case 3, when considered over sets of the correct domain.

4. For performance reasons when tackling certain problems.

Suppose that a problem has a number of possible solution strategies and a number of possible correct solutions, and that a machine with several processors is being used. Each processor should use a different strategy, and the first to yield a solution should cause this solution to be printed and the all processors to be stopped. Although there are perfectly good ways of controlling multiple processors in a deterministic manner [Hughes 84], this example seems to require genuine nondeterminism.

In practice it seems very difficult to add nondeterminism to functional languages without destroying many of their desirable qualities.

Referential Transparency

Functional languages bear a strong resemblance to the lambda calculus. Usually the only additions to the lambda calculus are a certain amount of syntactic sugaring, and perhaps the addition of some basic data items like integers and characters. For instance, consider the following program written in the language SASL [Turner 76]:

```
double n = n + n

double (double 3)
```

Although it may not appear so to a reader unfamilar with SASL, this program is really the same as any of the following lambda expressions:

(λdouble . double (double 3)) (λn. n + n)
(λn . n + n) ((λn . n + n) 3)
(λn . n + n) (3 + 3)
(3 + 3) + (3 + 3)
12

This close relationship with the lambda calculus that functional languages enjoy leads to many desirable properties, which I refer to collectively by the term *referential transparency*. Programs written in a functional language can be transformed into equivalent programs by applying very simple rules. For instance, consider the following three function definitions, provided in SASL and in a well-known imperative language:

```
fn1 x = myfn x + myfn x
fn2 x = 2 * myfn x
fn3 x = i + i WHERE i = myfn x
```

```
      INTEGER FUNCTION FN1(X)
      FN1 = MYFN(X) + MYFN(X)
      RETURN
      END

      INTEGER FUNCTION FN2(X)
      FN2 = 2 * MYFN(X)
      RETURN
      END

      INTEGER FUNCTION FN3(X)
      I = MYFN(X)
      FN3 = I + I
      RETURN
      END
```

The functions FN1, FN2 and FN3 above are probably the same, but there is no way of being sure without knowing something about the code for MYFN, which could contain updates or references to globals, input or output statements, or anything else for that matter. But the value returned by a call to a function in SASL is determined only by its arguments, and has no other effect except to return that value. In the SASL version of the example above we can be absolutely sure, without knowing anything about myfn, that fn1, fn2 and fn3 are identical and interchangeable. The importance of this property of functional languages is more fully described in [Turner 82] and [Darlington 82a].

Simple Input and Output

Functional languages are extremely powerful and succinct tools for expressing algorithms, but the limitations described above on the properties of functions seem to pose problems for input and output operations. For instance, it is not possible for a function to return the next value from an input stream, because it would need to return a different value every time it was called. It would not be possible for a function to print its argument at the terminal and return some useless value, for then the following two functions would not be interchangeable, as they should be:

```
fn1 n = print n + print n
fn2 n = x + x WHERE x = print n
```

The usual way in which simple input and output are modelled is by viewing the input and output streams as lists of data objects. For instance, consider the following interactive program. The input from the keyboard is represented as a

list of characters. The program evaluates to a list of characters, which are printed on the screen.

```
screen = map uppercase keyboard_input

uppercase x = ... (takes character x, returns an uppercased version)

map fn (a:rest) = fn a : map fn rest
map fn [] = []
```

(More details of the SASL language can be found in an appendix). uppercase is used as a high level function by map, which applies it in turn to every member of the list keyboard_input. Any function will do instead of map uppercase, I merely wish to demonstrate the principle that the output is purely and simply a function of the input. The result is that, for every character pressed on the keyboard, the uppercase version of it appears on the screen. This interactive program is a fairly trivial one, some more complex ones (built on the same lines) can be found in [Henderson 84].

If this program is to run interactively it is necessary to evaluate the program using *normal order* semantics. This is a scheme whereby values are only computed when they are needed to produce the next part of the output. Using normal order semantics, the value keyboard_input does not need to be fully evaluated before some output appears. Thus, a functional program can be used to describe an interactive activity.

Normal order semantics are usually implemented using *lazy evaluation* [Henderson 76]. This is a particular technique for implementing normal order semantics efficiently. It is important to note that, for many, lazy evaluation is a useful way of improving the performance of functional programs, by avoiding unnecessary work [Wadler 84]. But here it is being used to allow a functional program to perform some interactive task. The value keyboard_input in the program above represents what I shall call an "input stream", and is a very useful concept in trying to build functional programs that perform interactive tasks.

Lazy evaluation causes values in a functional program to behave in ways that many conventional programmers find counter-intuitive. For instance, a lazy functional list may seem to correspond closely to a data structure in a conventional program, but unlike the records of a conventional data structure the elements of a lazy list are not all constrained to exist at once. It sometimes helps to think of the lazy list as a partially formed data structure, with a closure (i.e. an unevaluated function) as its tail, which can be called when more elements of the list are needed. Under the scheme of lazy evaluation, when this closure is called it will yield one more element of the list structure, and form another closure which can be called when the rest of the list is required. Functional programmers often think of lazy lists as streams of values rather than as structures, and their use to represent communication with the outside world seems an entirely natural consequence of this.

Operating Systems

In [Henderson 82], Henderson expands the idea of using lists as input and output streams to deal with considerably more complex examples, including programs that handle multiple files, users and devices. To do this he introduces the idea of "tagging" and "untagging" lists of data items. A list of tagged data items is a list of (tag, data-item) pairs. A function tag turns a list of data items into a tagged list where all elements have the same tag. A function untag extracts from a list of tagged data items those items with a particular tag.

```
tag t (x:rest) = (t:x) : tag t rest
tag t []       = []

untag t ((t:x):rest)     = x : untag t rest
untag t ((wrong:x):rest) =     untag t rest
untag t []               = []

; thus, tag 5 [1, 2, 3]                 = [(5:1), (5:2), (5:3)]
; and   untag 5 [(5:1), (6:2), (5:3)] = [1, 3]
```

The use of these functions allows streams of values from several inputs to be combined into a single list. The elements of the list can then be processed by a function, and passed on to one of several outputs depending on where they came from. His approach has, in my view, a number of drawbacks.

> Another operation necessary: the merging of two (tagged) streams into a single stream. Simply taking alternate elements from each of the two input lists will not work: this operation will frequently be used on "input streams" from the outside world, in which case we want to take the next element from either list, whichever is ready first. This choice is nondeterministic.
>
> The second problem is one of style. When trying progressively larger examples the tagging, merging and untagging of multiple streams appears to form a web that is tangled and impenetrable, and the neat structure of the program rapidly disintegrates. The term "spaghetti programming" has been suggested to describe this. The operating systems of real machines involve a flow of data that is far more complex than that of the examples that Henderson tackles, and some experimentation indicates that this approach soon leads to confusion, with functional programming acting as a hindrance rather than as a help.
>
> All of the tagging and untagging leads to considerable extra work, if it is actually implemented with data manipulations as Henderson seems to suggest. Sometimes values need several layers of tagging, in order to get from one function to another, to which it is not directly connected. In a large, complex program, this could lead to considerable inefficiency.

Henderson's solution to the first problem is the introduction of a nondeterministic operator which I call `merge`.

```
almost_merge (a:rest) b  =  a : almost_merge rest b
almost_merge a (b:rest)  =  b : almost_merge a rest
almost_merge [] rest     =  rest
almost_merge rest []     =  rest

; e.g. merge [1, 2] [3, 4] = [1, 2, 3, 4]
;                        or   [1, 3, 2, 4]
;                        or   [1, 3, 4, 2]
;                        or   [3, 1, 2, 4]
;                        or   [3, 1, 4, 2]
;                        or   [3, 4, 1, 2]
```

merge behaves *almost* like the function defined above. If almost_merge were typed into a real SASL system, it would merely append two lists together: SASL systems try each clause of a function definition in turn, whereas what we want is to apply whichever of the first two rules can be applied first. merge takes two lists as arguments, and yields a single list containing all the elements of both. It does this (in current practical implementations) by creating a new process, and evaluating both of its arguments in parallel. Whichever produces a result first causes that result to be available to anyone requiring the value of this call to merge.

Now, this is not a function, and does not obey the rules for SASL functions that I described above. It does not have any side effects, but a given call to merge may return different results on different occasions with the same arguments. In particular, the expression

(λx. fn x x) (merge a b)

is not at all the same as

fn (merge a b) (merge a b)

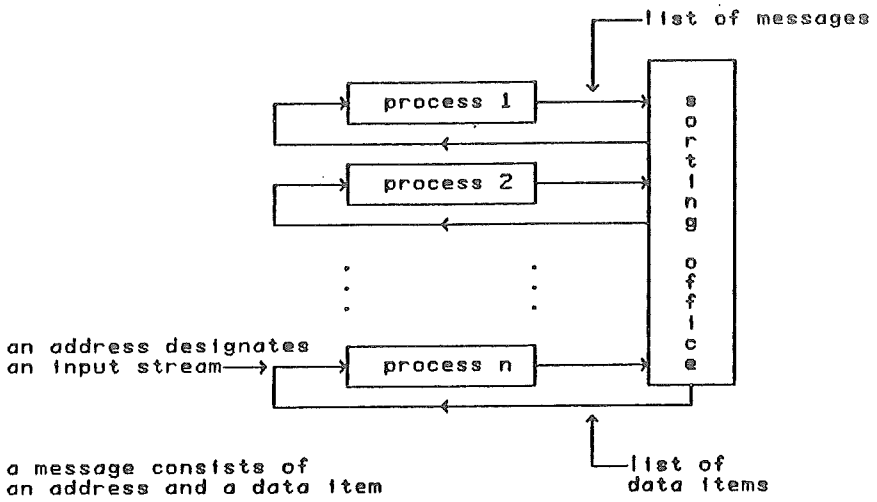
and so referential transparency is lost, and our ability to reason about functional programs is undermined.

Overview of the New Scheme

I present a new scheme for writing nondeterministic programs in a functional language. By avoiding the introduction of a primitive with a nondeterministic result (such as merge), the referential transparency of the language is retained. The idea of tagging streams of data items is retained, but large examples seem to avoid the "spaghetti" effect, and manage to avoid much of its inefficiency.

The scheme is based on evaluating a number of functional expressions in parallel. Each expression thus executed is called a *process,* and has a single input stream and a unique *process address.* Each process evaluates to a list of *messages,* which are pairs of the form (process address, data item). As each message is evaluated its data item is added to the input stream of the process to

which it is addressed.



The New Message Passing Scheme in Action

The body of each process is a function which takes a list (the input of the process) as its argument, and yields another list (the output of the process) as its result. These output lists are all lists of messages. The system evaluates all the processes in parallel, and acts like a sorting office for all the messages: each message consists of an address and a data item, and each address designates the input stream of some process. the data item in each message is sent to the input stream of the process to which it is addressed.

In Henderson's system `merge` may appear in any part of the program. This system constrains the use of nondeterminism so that it may only appear at the "bottom level" of a program. Programs written for it use no nondeterministic primitives or contructs, so the desirable mathematical properties of functional languages are retained.

A More Detailed Description

I now describe the operation of this system as a SASL program containting an occurrence of `merge`. This program would be terribly inefficient to run, but it serves as a very useful definition of the system as a whole. In the next section it provides the basis for a number of small example programs.

A simple case is described. The system has three processes, whose addresses are 1, 2 and 3. I have used numbers as addresses because later extensions to the system will allow an arbitrary number of processes to run, and so the use of more mnemonic identifiers seems inappropriate.

```
screen     = untag 0 messages
input_to_1 = untag 1 messages
input_to_2 = untag 2 messages
input_to_3 = untag 3 messages
messages = merge3 (start_1 input_to_1)
                  (start_2 input_to_2)
```

```
                    (start_3 input_to_3)
merge3 a b c = merge a (merge b c)
```

A system of three processes

It is informative to attempt to study the types of the various subexpressions of this program. I use a type notation that is somewhat informal, but based on that of Ponder [Fairbairn 82]. Capitalised words are "types", e.g. `Int`, or "type generators", eg `List[Int]`. An arrow denotes a function type, so that for instance the type of the integer addition function is `Int -> Int -> Int`. A double colon symbol means "is of type".

The screen object

```
screen = untag 0 messages
screen :: List[Char]
```

`screen` is the list of characters sent to the screen, and constitutes the "answer" in all of my examples. All messages sent to address `0` should have a character data item. When such a message arrives at the screen handler the corresponding character will appear on the screen.

The messages object

```
messages = merge3 (start_1 input_to_1)
                  (start_2 input_to_2)
                  (start_3 input_to_3)
messages :: List[Message]
Message = Pair[Address, Data_item]
merge3 a b c = merge a (merge b c)
```

`messages` is a list of all messages ever sent in the system, in the order in which they are processed. It is formed by merging together the output from the three active processes into a single list of messages, and then sending the data item in each message in this list to its destination. The term "output" is used in an informal way here, in fact it is the *value* of each expression which is considered.

The start objects

```
start_1 :: List[Data_item] -> List[Message]
start_2 :: List[Data_item] -> List[Message]
start_3 :: List[Data_item] -> List[Message]
```

These three functions are the bodies of the three processes. Each is applied to a value representing the input stream of that process The resulting values should be lists of messages.

Some Example Programs

1. A Simple Computation

```
start_1 in = tag 0 ("the answer is " ++ comp)
```

In this example we wish to perform a deterministic computation and display the result, as is possible with simpler systems. Only one process is necessary for this. The computation is called `comp` here and evaluates to a list of characters. The computation is performed yielding a list of characters, and this is converted to a list of messages by consing the destination onto each character. Thus `start_1`, when an argument has been applied to it, results in a list of messages. All of the messages are addressed to process `0`, and as a consequence will appear in the input to address `0`. Since address `0` designates the screen, the result of the computation will appear on the screen, preceeded by the characters "the answer is ".

The style of programming takes a little while to get used to. Note that the parameter of `start_1` is not used, but only because this example is very simple. It is necessary in order to ensure that `start_1` is of the correct type to be made into a process. Simple input and output are slightly more cumbersome than in other functional programming systems, but an enormous degree of flexibility has been gained, and as we show later the ability to hide the complexity of device drivers and input/output operations has been provided. This function may seem ugly and artificial, but the author of `comp` (the application program) does not have to know anything about it.

2. A Nondeterministic Computation

```
start_1 in =
  (0 : (hd in)) :
  (0 : (hd (tl in))) : []
start_2 in = (1 : 'a') : []
start_3 in = (1 : 'b') : []
```

`hd` and `tl` are functions that yield the head and tail of a list. `start_1` evaluates to a list of two messages, both of which are sent to the screen. `start_2` and `start_3` both send a single message, with a character data item, to `start_1`. The result of this program is to print either "ab" or "ba" on the screen.

The order in which operations occur when `start_1` is evaluated is very important. Process 1 evaluates to a list of two messages, whose data items are obtained by examining the process' input stream. Thus, the behaviour that is required of process 1 is as follows:

wait for an incoming message
send a message containing that incoming data item
wait for a second incoming message
send a message containing the second data item

Unfortunately, the program that I have specified so far will behave as follows:

send a message consisting of a closure, referring to the first
    data item on the input stream
send a message consisting of a closure, referring to the second

data item on the input stream

This is a consequence of the normal order semantics; the value of the data items is not required when the message is sent, so it will produce output messages (with unevaluated data items) before any input has arrived. The first interpretation requires the evaluating system to fully evaluate the data item on a message before it is sent, and in practice it's the preferable interpretation. In this particular example it makes no difference to the final result, but in many other cases it can lead to a message being sent before it is really ready. For instance, suppose that `start_2` and `start_3` were substantial calculations and that there were other possible processes that could send messages to the screen. If the second interpretation of the system's semantics were used then the calculations inherent in `start_2` and `start_3` would not be performed until the messages had already been accepted by the screen. This would congest the screen until those calculations had been performed.

Thus, the first suggested interpretation seems the best in almost all cases. The current system evaluates data items at the top level (but not completely, in the case of a data item consisting of a function or structure) before allowing them to be sent.

3. Input from the Keyboard

A convention will have to be decided upon for the behaviour of the keyboard device. We will assume that, as each key is pressed, the corresponding character is sent to process address `1`. A program to echo the characters pressed on the keyboard looks like this:

```
start_1 in = tag 0 in
```

Now, suppose that an application program wishes to take input from the keyboard a line at a time. Echoing and the delete key should be handled by "the system", the user program wants keyboard input to appear as a list of strings.

On functional systems that evaluate a single functional expression it is usually painfully obvious that this transformation (from a character based to a line based keyboard) is going on. It would typically be achieved by a function called something like `ch_to_line_kbd`, which takes a list of characters as input (the characters from the keyboard) and produces two outputs, the lines for the application program and the echoes to be sent to the keyboard. The use of functions of this sort rapidly leads to "spaghetti", and to data structures that persist and grow throughout the life of the program. This not only results in unclear programs, but if (for instance) it is necessary with file input and output, it could be extremely inefficient.

This program performs the necessary transformation under the new scheme in a painless way:

```
userprog line_kbd = ... (the client program, takes a list of strings
                         and yields a list of characters)

start_2 in = tag 0 (userprog in)

start_1 in = kbd_loop [] in
```

```
kbd_loop [] (DEL : cs)          = kbd_loop [] cs
kbd_loop (deleted : charbuf)
         (DEL : cs)             = (0 : BS) : (0 : ' ') : (0 : BS) :
                                  kbd_loop charbuf cs
kbd_loop charbuf (RET : cs)     = (0 : CR) : (0 : LF) :
                                  (2 : reverse charbuf) :
                                  kbd_loop [] cs
```

The function kbd_loop in this program performs the echoing and the building up of a line of input in a buffer. Its first argument is the character buffer, the second is the input stream to process 1. It handles keyboard echo and the delete key, and when return is pressed it sends a message containing a string to process 2.

The program may seem unpleasant at first sight, but it is painless in the sense that the writer of the user program may be unaware of its existence. Note that the systems programmer can implement whatever model he chooses for what to do about typeahead, buffering user output and so on, and can render his choices in a functional language.

(In the program, the identifiers DEL and CR are the codes generated by the delete key and the carriage return key. CR, LF and BS are assumed to be the control characters for carriage return, line feed and backspace. Delete is echoed as backspace, space, backspace and return as carriage return and line feed).

4. Polling the keyboard

How can a user program poll the keyboard, and react in different ways depending on the speed with which the user reacts? We will assume that the keyboard behaves as described in example 3. Process 2 now provides a service for the user process. It accepts two kinds of input messages: keys from the keyboard, and "has a key been pressed yet?" requests from users. It keeps a list of unclaimed keystrokes. This program is an example of a dialogue between two processes.

```
start_1 in = kbd_loop [] in

kbd_loop keys (ch : in)          = kbd_loop (keys ++ [ch]) in,
                                     character ch
; the first clause is only taken if ch is a character

kbd_loop (key : keys) (ad : in) = (ad : key) : kbd_loop keys in

kbd_loop [] (ad : in)            = (ad : []) : kbd_loop [] in

user_process kbdpoll self in = (kbdpoll : self) :
                               (0 : (null (hd in) -> 'x'; hd in)) :
                               user_process kbdpoll self (tl in)
; a -> b; c means "if a then b else c"
```

The function kbd_loop runs as process 1, while the user program is process 2. kbd_loop behaves in three different ways, depending on the characters sent to it:

If a character arrives, add this to the list of unclaimed characters.

If a request arrives and we have characters saved up,
send the first character to the requester
If a request arrives an we have no characters saved up,
send a nil value to the requester

These are reflected in the three clauses of the function definition.

The result of this example is a continuous stream of "x"s on the screen, interspersed with the echo of the keys pressed on the keyboard when they occur. More complex protocols will allow any type of interaction between processes, based on question-and-answer type dialogue.

Creating New Processes

The previous section demonstrated a system involving a fixed, finite number of processes communicating with each other via messages, and showed how such a system could be used to build a number of interesting programs. In the examples I have tried to show how it leads to better organised and more efficient programs than the use of merge operators. In this section a mechanism is presented for allowing such a network to change dynamically, so that new processes can be created as required.

The creation of new processes works as follows. Each process executing has a unique process address, to which messages may be sent. In addition to the addresses of the processes executing there may be other addresses to which messages can be sent, causing various special effects. One of these is the screen: we have declared that any message sent to this address must have a character data item, and that sending such a message will cause the character to appear on the computer's video screen.

In a similar way, there is another special address called the *process creator*. Any messasge sent to this address must have as data item a function of a suitable type (which is described more precisely below). Sending a message to this address causes a new process address to be created, and starts up a new process at that address using the data item of the message as its body.

This mechanism makes simple examples rather harder to understand. Because processes are created dynamically, so too are process addresses. When a new process is created its address must somehow be given to everyone who will want to send messages to it.

This operation is described with more detail and precision by the following program. Address 0 is the screen, and 1 the process creator. There is now only one initial user process, whose body is constructed from the function `start` and whose address is 2. More processes can be created as required.

```
screen = untag 0 messages
process_bodies = start : untag 1 messages
messages = MERGE (apply_to_each process_bodies (2...))
```

A system supporting an arbitrary, dynamic number of processes

The screen object

```
screen = filterout 0 messages
screen :: List[Char]
```

screen serves the same purpose as before, it is the "answer", or output, of the whole scheme. Note that in a more complex case there could be many other devices and special addresses visible at this level, depending on the hardware configuration involved. This is the level at which all such things get bound together.

The list of process bodies

```
process_bodies = start : untag 1 messages
process_bodies :: List[Process_body]
Message = Pair[Address, Data_item]
Address = Int (in these examples)
Data_item = any type
Process_body = Address -> List[Data_item] -> List[Message]
```

process_bodies is the list of all process bodies that ever exist, in the order in which they are created. This shows the precise type that is expected of an object sent to the process creator, and how the new address is made accessible to the program. The new process body has applied to it its own (newly created) process address, and its own (newly created) input stream. It should then evaluate to a list of messages. If this list is exhausted then the process is "dead" (messages sent to it will have no effect).

The list of all messages

```
messages = MERGE (apply_to_each process_bodies (2...))
messages :: List[Message]

apply_to_each (body : bodies) (i : ads) =
  body i (untag i messages) :
  apply_to_each bodies ads
```

This is a list of all messages ever sent in the system, in the order in which they are processed. It is constructed by merging together all the members of a list of lists of messages. Each of these lists is the output of a process. Each process constructed by applying to a process body (which has been sent to process 1) a new, unique address and an input stream. The 2... is an infinite list of integers (i.e. the value [2, 3, 4, 5, 6, ...]) and is used as the generator of unique addresses. Each body is applied to its own address (i.e. a value taken from 2...), and the list of data items from all messages that are addressed to it (this is what the untag 1 achieves).

MERGE (read as "big merge") takes a list of lists as its input, and nondeterministically merges all elements of these lists together into a single list.

This feature is included in the system currently being used on SKIM: in fact, it was only recently that it occurred to me *not* to use it. It would be instructive to explore more fully the use of the system without this feature: a static collection of processes seems much simpler, and could form the basis of the software for a

machine with many processors. Rather than separating "processes" and "processors" in the underlying machine, we force these to correspond exactly. It is the responsibility of the functional programs to distribute work in a useful manner. The bookkeeping involved in managing "processes" seems to be one of the tricky aspects of multiprocessor design, which the use of such a scheme would avoid.

Other Possible Extensions

Control over Scheduling

The current scheme runs on a single processor, and will run any process that is eligible. It is not possible to implement a background task, whose scheduling is under more precise program control.

One possible method for implementing background tasks would be to designate one process as "background", and only run it if nothing else can. Alternatively, some more complex scheme could be implemented in the underlying system giving different priorities to the different tasks, and including a scheduling algorithm based on these priorities.

A more powerful idea is to use the "engine" concept [Haynes 84]. A "background" process is created in a special way that marks it as such. It may only compute when a special "clock tick" message is sent to it. Other processes comprising the operating system scheduler decide when some time can safely be spent evaluating this background task and send a "clock tick" message to it when this is the case. This is effectively permission to perform a small amount of computation on this process, after which the scheduler should be run again and may decide what to do. This scheme allows more precise control, for instance it might be appropriate if multiple background tasks with complex priority functions were required.

Another aspect of this is the use of the same system on a machine with several processors. Although processes are able to pass objects (or functions) of any type between them, there is no reason why they should all be running on the same processor, or even in the same address space. By making the process creation operation more complex it should be possible to write a system that runs on a machine with several processors. The allocation of processors to processes could be explicit, or performed by the system. One of the most pleasing aspects of the scheme at the moment is how little it assumes about the computer on which it is being run.

Error Recovery

This is very tricky, as there are a number of ways in which a user program could go wrong, even if it has been proved "correct".

The simplest case of this is a value error detected at runtime, such as an attempt to divide by zero. Saying that the result is "error" is not really adequate: some method is needed of telling the operating system that something has gone wrong, and allowing the system to decide what to do about it.

The next example is an infinite loop, or a long computation that the user realises that he did not want. The user signals impatience by hitting break, or signalling in some other way that the attempt to evaluate this object should be

abandoned. In this case the task should be killed - how is this to be achieved? There seems a need for a "kill-process" primitive. (This is also necessary to tackle case 4 in the introduction, where nondeterministic behaviour is being used to perform a breadth-first search). This would be implemented as a "death" message, a special data object which, when sent to a process, removes that process from the system.

Even worse that this is a program that uses up all of store: an accidental attempt to reverse an infinite list will do this. The machine must decide to throw something away before it can allow any more functional programs to run, as functional programs always need some spare heap in order to compute. In this case it seems necessary to mark in some way which processes are "untrusted". If a store jam occurs, the machine throws away all untrusted processes and sends a message to some agreed location to say that this has occurred.

What if this does not release any store? A user program creates a large structure, and passes this to a trusted process, who unwittingly holds on to it. This would prevent the garbage collector from reclaiming it, and the machine would die for lack of store. Preventing this seems to require some care. One method for presenting this is to keep a small reserve of store, which the user cannot use up. This is only released when the machine is diagnosed as being full. However, this will cause a reduction in processing power (because heap based machines go faster if more free memory is available). Another idea is to control objects passed from an untrusted program, and ensure that they are all fully evaluated, and of controlled size. The user program is a function rather than a process body, and so rules like this can be successfully enforced.

It is worth noting that recovery seems even more complex from certain errors that involve multiple processes. For instance, a program that went wrong by generating thousands of superfluous processes until the machine filled up might be hard to recover from. Perhaps this would be solved by making the processes created by an untrusted process also untrusted. More thought is needed in this area.

Most user programs are deterministic, but some may not be. For instance, the ability to run experimental versions of the operating system under the old one would be extremely desirable. This may require a more complex version of the "trust" mechanism, but would almost certainly be worth it.

Operating System Design

The new scheme is currently being used to write an operating system for SKIM, a microcoded processor specifically designed for the execution of functional languages. The intention is to make SKIM a self-supporting computer system, on which functional programs can be edited, compiled and run: for this, a simple operating system is required. Although this operating system is still under development, some discussion is worthwhile on the ways in which this scheme influences the design of the operating system.

The result is not, sadly, an explosion of new ideas on how operating systems should be designed. But there is a clear impression that it is possible to do things that conventional operating systems are able to do, and with no significant loss of performance. I believe, although there is no evidence for this yet, that

such an operating system can be constructed using clearer and simpler programs than their conventional counterparts in a conventional operating system. If this is the case it will be a considerable victory for functional programming.

Process Control

The kernel of an operating system is frequently thought of as that part which controls processes and scheduling: if this is the case, to what extent has SKIM cheated by implementing this sort of thing in microcode? In answer to this it seems worth pointing out that the model of a process implemented here is very much less complex than the processes implemented in a conventional operating system. The processes in this scheme provide a basic mechanism for controlling parallel execution, and are unlikely to correspond to what the user thinks of as tasks being performed by the machine. Such ideas as who the process belongs to, what the process is called and what resources it is allowed to access must be organised by other programs. Many fundamental decisions concerning the structure of the operating system are still to be made, and will be expressed in a functional language.

The Filing System

There is no reason why a filing system should not be built along much the same lines as in conventional operating systems. At the lowest level programs would communicate with the disc through messages, which read and write sectors on the disc. Further layers on top of this could control file structure, the garbage collection of disc blocks, archiving, directories, version control and so on.

Alternatively, the storage on which the filing system is based could be available over a local area network. Using similar communication protocols the functional program can deal with such a situation in exactly the same way that a conventional machine on a network does.

It would also be possible to use the disc to implement a virtual memory system, which retains the integrity of the heap when the system is powered down. This would allow objects to be "filed" simply by holding on to them as data structures: a large virtual memory would ensure that they eventually migrated onto disc, and the result is, to a far greater extent, a "functional" program. My instinct, however, is to be wary of this approach for the following reasons:

1. It demands far more implementation effort "beneath" the level of functional programming. It is the low level parts of operating systems whose semantics are most in doubt and where hidden bugs are most harmful: this is where functional programming is needed most!
2. It seems more liable to disaster from some hidden bug in the operating system. Merely discarding a pointer might discard the filing system, requiring recovery in some manner "outside" the operating system.
3. It merely delays the issue of how one is to communicate with a persistent outside world. Sooner or later the question of interfacing to networds, remote filing systems and other computers must be faced.

How a filing system *should* be represented in a functional world, and the application of such an approach to a networked world (with the functional program being distributed over the network), are still the subjects of debate and research in the functional programming community. But it is pleasing to note

that the use of this scheme provides a wide range of choices, and that one is not limited by problems of efficient representation. In the particular case of the SKIM machine, it seems most appropriate to stick to a conservative model of what a filing system is.

The User Interface

The design of the user interface is not forced by this scheme in any way at all. For instance, if desired a shell program could be written, along the lines of the UNIX shell. This could take input from the keyboard and access the filing system in order to perform tasks specified by the user. If required, it could maintain a history of previous commands, and any other fancy features that are required. Alternatively a menu scheme could be used.

It is amusing to note that the commands given to a shell effectively form an imperative command language, and there is no reason why programs should not be written by the user in this language. Hardly functional programming! Somehow, it should be possible to find something a bit more functional: I welcome suggestions. When sitting at a terminal giving individual commands to a computer, the imperative paradigm really does seem the most appropriate. It may be that this tendency can be minimised by providing a good menu-driven interface, and a good collection of basic commands.

Some thought concerning commands available under the UNIX shell reveals that many of them are *almost* functions: the construction of pipe expressions, for instance, is very much like functional programming. In the command language of a functional machine this style of programming would be heartily encouraged, with the considerable bonus that such a style would not be constrained to work merely with streams of characters. By doing a certain amount of type analysis, as is performed in many of today's functional languages, it should be possible to provide considerable expressive power using this style of command language.

Part of the (highly successful) UNIX philosophy is to think of programs as tools, allowing many common tasks to be performed simply by combining existing programs using a functional style. This benefit is limited to programs that communicate using streams of characters, and is distinct from the provision of subroutine libraries for programming languages. By encouraging the use of a functional style at every level (i.e. in the writing of applications and tools, and in the command language) programming tools become even more general. We approach a state where, if a programming task has been solved, the solution of the problem is available for re-use at the command level and at the language level.

Areas of Uncertainty

The scheme presented here grew from a practical requirement, the need for an operating system on a real machine that could only be programmed in a functional language. An acceptable mathematical characterisation for it has not yet been found. It may have many theoretical ramifications that have not yet been fully explored.

What are the semantics of this new system?

The semantic description of programs written under this scheme appears just as hard as in a program involving nondeterministic merges. Are the programs easier to control and understand if referential transparency is retained? The system without the process creation mechanism may be considerably simpler. What are its limitations for the writing of practical systems?

How useful is type checking in this regime?

I have glossed over some of the problems concerned with type checking in the discussion above. If the items taken as input by a process form a list then all objects sent to one particular process should all be of one type. This may be inconvenient in some circumstances, but it is still perfectly possible. However, a process can send messages containing data items of different types, provided that each one is of an appropriate type for its receiver. So what exactly is the type `Message`? Current type checking technology seems unable to describe it. Under such a scheme an `Address` would be a type generator rather than a type, so that `Address[T]` is a process address to which objects of type `T` should be sent.

```
Message = Pair[Address[T], T] for some T, rather than for all T

; e.g. if    intad :: Address[Int]
;      and   chrad :: Address[Char]
;      then  [(intad : 3), (chrad : 'a')] is a legal list
;                                              of messages
```

Current languages seems unable to describe this type

The type system described in [McQueen 84] can describe this type, but no mechanical checker for such types exists. SKIM's operating system is in fact written in Ponder, a language with a type system that is somewhat more powerful than that of ML [Milner 78], but still unable to describe the type `Message`. SASL has been used in examples here because it is more widely known and more succinct. However, there are numerous places in the operating system where the type-checking is sidestepped. It seems a shame that this should be necessary.

It's in a functional language, but is it functional programming?

The programs presented here have all been written in a functional language, yet they display behaviour that is not normally associated with such programs. Using processes "side effects" can easily be manufactured if desired, based on the ability of a process to have a "state" which other processes can interrogate or update. How will this affect large programs if such facilities are available? It may be that programmers will be tempted to user methods just as bad as any conventional program: this remains to be seen.

Functional programming has always been difficult to define precisely, and it may be that, although concerned with functional languages, this scheme really steps outside the realm of functional programming as it is generally accepted. Programs are still functions, but their structure is quite complex. The value of expressions under this scheme is still deterministic, but the scheme does more

than just take the value of an expression. On the other hand, something has to be done if functional languages are to be used to describe and implement time-dependent and nondeterministic systems. This scheme represents an attempt to achieve just that.

References

[ACM 84]:
Proceedings,
1984 ACM Symposium on Lisp and Functional Programming,
Austin, Texas,
August 1984.

[Darlington 82]: J. Darlington, P. Henderson and D. A. Turner (eds),
Functional Programming and its Applications,
Cambridge University Press,
1982.

[Darlington 82a]: J. Darlington,
Program Transformation,
pp. 193-215 of [Darlington 82].

[Duce 84]: D. A. Duce(ed),
Distributed Computing Systems Programme,
Peter Peregrinus Ltd.,
IEE Digital Electronics and Computing series 5,
1984.

[Fairbairn 82]: J. Fairbairn,
Ponder and its Type System,
Cambridge University Computer Laboratory Technical Report #31,
1982.

[Fairbairn 84]: J. Fairbairn,
A New Type Checker for a Functional Language,
Cambridge University Computer Laboratory Technical Report #53,
1982.

[Henderson 76]: P. Henderson, J. H. Morris,
A Lazy Evaluator,
3rd Annual ACM POPL,
1976.

[Henderson 82]: P. Henderson,
Purely Functional Operating Systems,
pp. 177-189 of [Darlington 82].

[Henderson 84]: P. Henderson, S. B. Jones,
Shells of Functional Operating Systems,
pp. 290-298 of [Duce 84].

[Hughes 84]: R. J. M. Hughes,
Parallel Functional Languages Use Less Space,
Programming Research Group,
Oxford University,
1984.

[Haynes 84]: C. T. Haynes, Daniel P. Friedman,
Engines Build Process Abstractions,
pp. 18-24 of [ACM 84].

[MacQueen 84]: D. B. MacQueen, Ravi Sethi,
An Ideal Model for Recursive Polymorphic Types,
11th Annual ACM POPL,
1984.

[Milner 78]: R. Milner,
A Theory of Type Polymorphism in Programming,
JCSS, 17(3), pp. 348-375.

[Stoye 83]: W. R. Stoye,
The SKIM Microprogrammer's Guide,
Cambridge University Computer Laboratory Technical Reprot #40,
1983.

[Stoye 84]: W. R. Stoye, A. C. Norman and T. J. W. Clarke,
Some Practical Methods for Rapid Combinator Reduction,
pp. 159-166 of [ACM 84].

[Turner 82]: D. A. Turner,
Program Transformation,
pp. 193-215 of [Darlington 82].

[Turner 76]: D. A. Turner,
SASL Language Manual,
University of St. Andrews Computer Science Department,
1984.

[Turner 79]: D. A. Turner,
A New Implementation Technique for Applicative Languages,
Software Practice and Experience, Volume 19, pp. 31-34, 1979.

[Wadler 84]: P. Wadler,
Listlessness is Better then Laziness:
Lazy Evaluation and Garbage Collection at Compile Time,
pp. 45-52 of [ACM 84].

Appendix 1

The SASL Language

SASL [Turner 76] is a simple, succinct functional language. It is strongly related to KRC [Turner 82] and Miranda [Turner ?]. This guide is intended not as a complete language reference manual, but as an aid to those seeking to understand the examples in the previous text.

Functions

The main operation for combining values in SASL is function application, which is denoted by juxtaposition. So, the expression

```
fn x
```

has a value that is obtained by applying the value `x` to the value `fn`. Function application associates to the left, and brackets may be used to indicate grouping. So,

```
plus x 3
```

is the same as

```
(plus x) 3
```

and denotes `x` added to 3 (assuming that is what the function `plus` does). All functions are fully curried, i.e. arguments may be applied to them one at a time. Functions may be used as first class data objects. The value that a function yields is uniquely defined by the arguments passed to it, and two textually identical expressions (in the same context) will always yield the same value. There are no "side effects" or assignment statements.

Primitive Data Types

All objects in a SASL program are integers, characters, pairs or functions. The language has no type checking scheme, so type errors may occur at run time and will presumably produce some message of complaint from the language system.

SASL closely resembles the lambda calculus in its semantics, but in order to make simple operations easier to understand some primitive data objects have been added. These are characters, integers and pairs. Character constants can be included in a program as the intended character in single quotes. Integer constants are a sequence of digits, with an optional leading minus sign. Pairs are written using an infix colon.

```
'A' 'B' '$'          character constants
23 128 15            integer constants
'a' : 23             a pair consisting of a character
                                    and an integer
1 : 2 : 3            read as (1 : (2 : 3))
```

A list is a sequence of pairs, terminated with the "nil" value, which is written as [] . A list may be written as

```
[1, 2, 3, 4]
```

which is a shorthand for

```
1 : 2 : 3 : 4 : []
```

In addition, string quotes may be used to denote a list of characters. So,

```
"hello"
```

is a shorthand for

```
'h' : 'e' : 'l' : 'l' : 'o' : []
```

Expressions

There are a number of predefined primitive functions and infix operators in the language. For instance,

```
+ - * /          simple arithmetic operators
hd tl            operations to extract the left and right parts
                                        of a pair
++               append lists
```

Function Definition Clauses

A function is defined by a sequence of one or more "clauses". For instance,

```
double x = x + x
```

is a clause which gives a value to the name double. A clause consists of the name to be defined, followed by zero or more arguments, followed by an equal sign, and then the corresponding expression. Thus, double can be thought of as the lambda expression

$(\lambda x.\ x + x)$

Arguments can be patterns instead of simple names, indicating that "this clause is only appropriate if the actual argument matches the pattern". This can lead to the need for more than one clause in the definition of a function name. For instance,

```
append (a:x) y = a : append x y
append [] y = y
```

This function append appends two lists together. The first clause only applies if the first argument to append is a pair: if it is, the body of the clause is the result

of the call to append, using a and x to denote the left and right parts of the first argument. The second clause only applies if the first argument is nil. When evaluating a call to append, each clause is tried in turn until one matches: this is then taken to define the value returned by the call.

Note that recursion may be used freely.

Note that argument passing is a way of giving names to subexpressions.

Another method of doing this is the WHERE construct. This takes the form

```
<expression> WHERE <sequence of clauses>
```

allowing a large expression to be broken up into a number of smaller ones. For example,

```
fn a b c = (x1 / quot) : (x2 / quot)
                WHERE x1 = b + d
                      x2 = b - d
                      quot = 2 * a
                      d = b * b - 4 * a * c
```

High Level Functions

Functions can be manipulated as values in the language. This extremely powerful facilitiy replaces the need for many looping and other control constructs. The functions map and fold illustrate this:

```
fold op s [] = s
fold op s (a:x) = op a (fold op s x)

map fn [] = []
map fn (a:x) = fn a : map fn x
```

map takes two arguments, a function and a list. The function is applied to every member of the list, and the result of the call to map is another list consisting of the results of these calls. These small examples show typical uses of map:

```
map double [1, 2, 3]          is   [2, 4, 6]

plus a b = a + b
map (plus 3) [1, 2, 3]        is   [4, 5, 6]

map (map double) [[1, 2],     is  [[2, 4],
                  [3],             [6],
                  [4, 5]]          [8, 10]]
```

The second and third examples show how arguments may be passed to a function one at a time. For instance, plus in the second example is a function that takes two integers and yields another integer. plus 1 is a function that takes one

integer and yields another. In a similar way, `map (map double)` is a function that takes a list of lists of integers and yields another list of lists of integers.

The function `fold` is slightly more complex, and the best way to understand it is to try it on a few sample arguments.

```
fold plus 0 [1, 3, 12]              equals    16  (the sum of the list)

fold append [] [[1, 2], [3], [4, 5]]  equals  [1, 2, 3, 4, 5]
```

`fold`'s first argument should be a dyadic function (a function expecting two arguments), and its third argument should be a list. The function is applied "between" the elements of the list.

Appendix 2

An Efficient Implementation

The SKIM machine is a microcoded combinator reducer. In addition to a reducer, the microcode also includes a timeslicing mechanism which implements the scheme described in this paper. This appendix describes how it works at the machine level, and may be of interest to other implementors. The scheme is described for a combinator reducer, but is equally applicable to any other form of lazy evaluator.

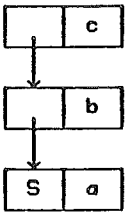
The Heap

SKIM's memory is organised as a heap of Lisp-like "nodes". Each node contains two "values", the "head" and the "tail" of the node. Values consist of a 20 bit "data" field and a 4 bit "tag" field. The tag can take one of the following values:

value is	abbreviation	data is
application	appl	heap pointer
pair	pair	heap pointer
process address	addr	heap pointer
integer	int	value
character	char	ASCII code
nil	nil	0
combinator	comb	microcode address

Types of value on the SKIM machine

For instance, the value S a b c is represented as:

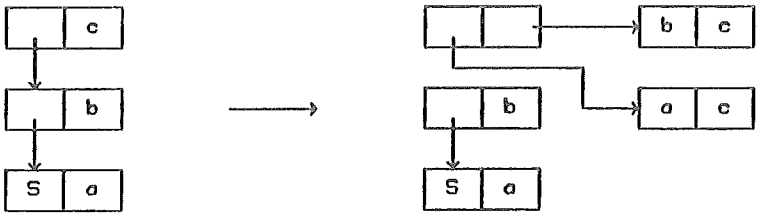


The representation of S a b c in SKIM's memory

The store is organised as a heap, and the garbage collector uses the tags on all values to determine which parts of memory are accessible and which are garbage. There are also other bits on each cell in memory which are used by the garbage collector to arrange this.

The Reducer

The bulk of SKIM's microcode is a normal order graph reducer, as described in [Turner 79, Stoye 84]. It uses pointer reversal to crawl over a graph of application pointers, which is continuously transformed until it will not reduce any further. For instance, if given the expression S a b c the reducer would perform the following transformation:



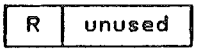
S a b c is transformed into a c (b c)

Conventional reduction of the S combinator

Thus the result of the reduction is a c (b c), as we expect.

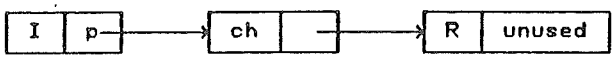
The Input Stream

In order to implement input streams, a special combinator called R (for "read") is used. The input list's value is initially represented as an application pointer to:



An input list's value is an application pointer to a cell like this

The keyboard system also keeps a pointer to this cell, which is known as the "readin" cell. When a key is pressed, this cell is overwritten like this:



⟶ is an application pointer

p⟶ is a pair pointer

The readin value after a key has been pressed

The cell on the left used to contain an R

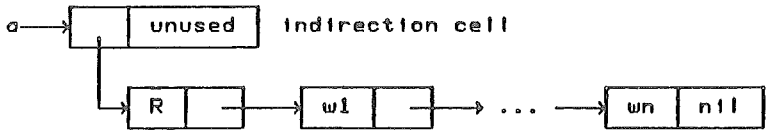
The I cell is necessary because the initial cell will have application pointers to it. The second cell is a pair cell, and the readin cell (where the next character will appear) has advanced to being the cell on the right.

The Slicer Data Structures

The microcode that controls the scheduling of the processor is called the "slicer". It keeps a circular queue of "active" processes, i.e. those that may be evaluated. Each process is represented in this list merely by its value. Each process also has an input stream, at which messages sent to it appear. If an attempt is made to evaluate a readin cell then the running process will wait until something is sent to that readin cell, Thus the process is removed from the list of active processes and added to the list of processes waiting for something to arrive at this input cell.

Note that the value of a process (i.e. the stream of messages output from the process) and its input stream are not bound together in any way. Because partially unevaluated messages can be sent, there is no reason why a process should not end up attempting to evaluate the input stream of another process, indeed this sometimes happens in real programs. An arbitrary number of processes can be suspended, waiting for input at one readin cell.

For this reason each input cell is represented by the following arrangement:

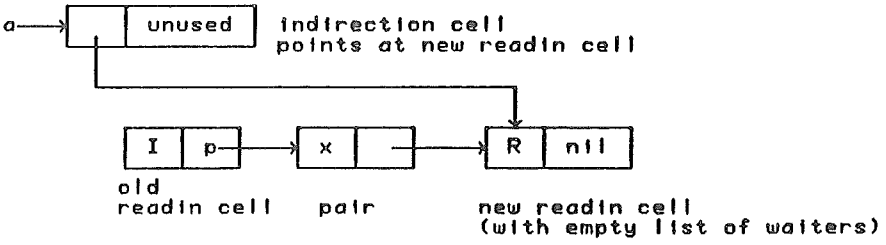


——→ is an application pointer
p——→ is a pair pointer
a——→ is an address pointer

readin cell list of processes waiting on this readin cell

The full structure of a readin cell

The tail half of the read cell holds the list of those waiting for input at this cell. The process address is represented by pointers to an indirection cell. If a message with data item x is sent to this cell, all the waiting processes will be added to the active queue and the readin structure will be turned into:



——→ is an application pointer
p——→ is a pair pointer
a——→ is an address pointer

A readin cell after a message send

The Action of the Slicer

The slicer runs the reducer on the process on the head of the queue of active processes. The process should evaluate to a list of messages. It evaluates the first list cell, the cell representing the first message, the head of that cell (i.e. the destination address), and the tail of that cell (i.e. the message's data item). This action is performed until:

The process evaluates to nil:
The process is removed from the active queue: it is dead. The slicer continues by evaluating the next active process.

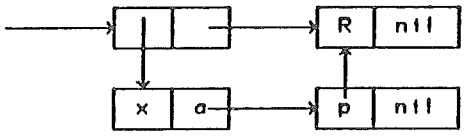
5000 reductions have been performed (i.e. an arbitrary time limit):
The slicer advances to the next member of the active queue. This is to prevent a process preforming intensive computation from locking the others out.

The evaluator attempts to evaluate an R combinator:
Further reduction of this value is dependant on a message arriving at the readin cell. This process is removed from the active queue and added to The slicer continues by evaluating the next active process.

The process evaluates enough to produce a well formed message:

This is equivalent to sending a message. The message will have two parts, a destination address (fully evaluated) and a data item (evaluated at top level). The process value in the active queue is overwritten with its tail, so that futher reduction on this process will yield the next message in the list. Unless the destination address is something special, the data item is added to the input stream indicated by the destination address, and all waiters are removed from that readin cell and added to the active queue. The slicer proceeds by evaluating the next active process.

Special destination addresses include any devices, and the process creator. The action for devices is device dependant but not hard to decide upon. Note that the evaluator may have to poll input devices and stop so that they can send their input messages in some way. The action for the process creator, when sent a process body `x`, is to add the following structure, as a process to be evaluated, to the active queue.



The structure of a new process using process body x

——→ is an application pointer
p——→ is a pair pointer
a——→ is an address pointer

The result is added to the active queue as a process to be evaluated.

Discussion

The mechanism has a pleasing feel of being minimal in some sense: there are very few arbitrary choices in its design, and no special cases or features. It is reasonably fast, and has no gradual space leak. However, there are some points that need more thought:

1. The SKIM reducer reverses pointers while reducing an expression, and a process switch requires any existing chain of pointers to be re-reversed, so that the new running process does not fall foul of the reversed pointers of the other processes. Thus some task switches may take rather a long time. This happens when waiting for a message, and when the time limit comes into effect. Removing this overhead seems quite hard, and would probably need considerable changes to the reduction method.

One possibility is to mark in some way cells that have reversed pointers in them (at the moment it is not possible to detect them). If a process meets a reversed value then it and another process both need some common subvalue, but the other process has already started evaluating it. So add a "branch" to the reversed chain and suspend the current process, in such a way that it is restarted when the other process un-reversed this list. In a conventional system this is unlikely to be worth the effort, but arrangements of this sort might be quite useful in a multi-processor machine.

2. Ensuring fairness needs a certain degree of care and compromises response time. This is the requirement that no combination of processes should be able to grab all CPU time by sending messages to each other. When choosing which process to run next, it is best to choose one which has not been involved in the last transaction, in case two processes (by furiously sending messages to each other) manage to lock all the others out. This means that, when a message is sent, the activated process should not be the next one to run: even if it is probably the one who should run next if response time to devices is to be improved.