Science of Computer Programming 6 (1986) 291-311
North-Holland

MESSAGE-BASED FUNCTIONAL OPERATING SYSTEMS

William STOYE
Computer Laboratory, University of Cambridge, Cambridge CB2 3QG, United Kingdom

Communicated by J. Darlington
Received February 1985
Revised May 1985

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.

1. Introduction

The work described in this paper was developed in a practical setting. The SKIM machine is a microcoded combinator reduction machine developed at Cambridge University and described in [16]. It can be used as the basis for very high performance implementations of functional languages that are based on the normal order lambda calculus (e.g. SASL [17], KRC [19], Ponder [5] and Miranda [20]). Having constructed it, the next task is to turn it into a usable machine that can support editors, compilers and the like. This means that device handling, filing systems, error recovery and so on have to be considered, and leads to the need for a new understanding of how to control inherently non-functional operations from within a functional program. The implementation of the resulting model as part of SKIM's combinator reduction software is described in Appendix A, the rest of this paper describes it as a more abstract model.

2. Simple input and output

Functional languages are extremely powerful and succinct tools for expressing algorithms [3], but the way in which the results of such calculations should be communicated to the outside world is not obvious. For instance, it is not possible for a function in a functional program to return the next value from an input stream, because it would need to return a different value every time it was called. It is not 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
```

All example programs in this paper are written in Miranda, of which a short summary can be found in Appendix B. The interchangeability of the two functions above is a consequence of the *referential transparency* [19] of functional languages.

The usual way in which simple input and output is modelled is by using lists of data objects to represent input and output streams. For instance, consider the following interactive program. The input from the keyboard is represented as a list of characters. The program evaluates to another list of characters, the members of which are printed on the screen.

```
screen = map uppercase keyboard_input

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

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

uppercase is used as a high level function *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 a pure 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 inteactive program is a fairly trivial one, some more complex ones (built on the same lines) can be found in [8].

3. Operating systems

In [7], 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 onto one of several outputs depending on where they came from. His approach has, in my view, a number of drawbacks.

A further operation is 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. (This is the least important objection, if the use of *merge* were the right approach on the grounds of style then this problem could certainly be overcome with some effort.)

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 as (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 Miranda system, it would merely append two lists together: Miranda 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. Its evaluation does not cause any side effects, but a given call to merge may return different results on different occasions with the same arguments. In particular, the expression

$$(\lambda x.fn\ x\ x)\ (merge\ a\ b)$$

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.

4. Message passing

The scheme of Fig. 1 represents a new approach to 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.

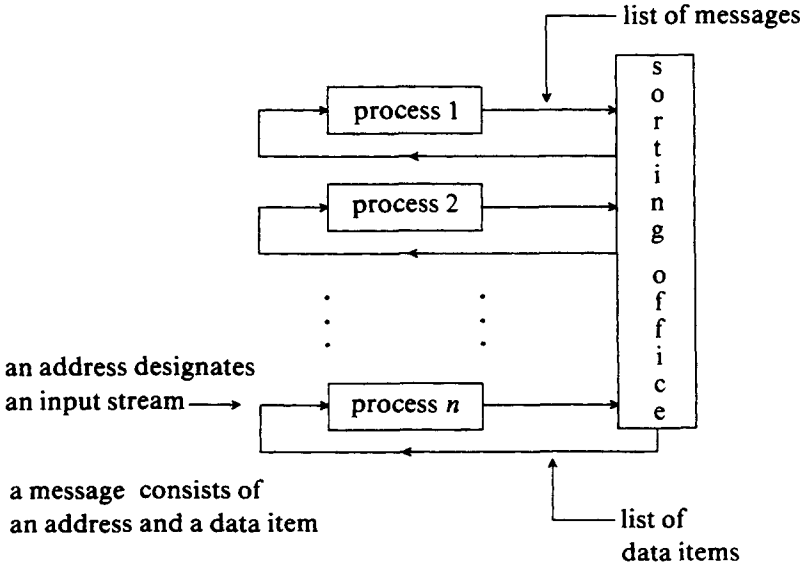


Fig. 1. The new message passing scheme in action.

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.

Because the input streams are represented by lazy lists, there is in effect an arbitrary buffer on each input stream. The action of 'reading' an incoming message is not well defined, and not synchronized in any way.

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 constructs, so the desirable mathematical properties of functional languages are retained.

5. A more detailed description

I now describe the operation of this system as a Miranda program containing 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 (see Fig. 2). 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)
```

Fig. 2. A system of three processes.

It is informative to attempt to study the types of the various subexpressions of this program. I use Miranda's type notation, with the convention that capitalized words are types or type generators. The following declarations introduce some of

the symbols used:

```
Int ≡ num          ; example of type identity—num is the
                   ; built in Miranda
                   ; name, I prefer capitalized types
Char ≡ char        ; synonym for built-in Miranda name
String ≡ [Char]    ; a string is a list of characters
plus :: Int → Int → Int  ; a double colon means 'is of type'
```

The screen object

```
screen = untag 0 messages
screen :: [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 :: [Message]
Message ≡ (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 :: [Data_item] → [Message]
start_2 :: [Data_item] → [Message]
start_3 :: [Data_item] → [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.

6. Some example programs

6.1. 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, which 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, preceded 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: the vast majority of applications programmers will never use message passing, and need have no understanding of its operation. The examples shown here are mainly of relevance to the authors of operating systems.

6.2. A nondeterministic computation

```
start_1 in = [(0, hd in), (0, hd (tl in))]
start_2 in = [(1, 'a')] ; i.e. a list with one element,
start_3 in = [(1, 'b')] ; which is a pair
```

`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.

6.3. 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 screen. 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 user's program, of type [String] → [Char]
start_2 in = tag 0 (userprog in) ;  this task runs the user's program
start_1 in = kbd_loop [] in ; this task is the keyboard hander
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 can 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).

6.4. 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 1 provides a service for other processes. It accepts two kinds of input: keys from the keyboard, and "has a key been pressed yet?" requests from users.

```
start_1 in = kbd_loop [] in

kbd_loop keys          (ch:in) = kbd_loop (keys++[ch]) in,
                                 character ch
kbd_loop (key : keys)(ad:in) = (ad, key) : kbd_loop keys in
kbd_loop []            (ad:in) = (ad, nullchar) : kbd_loop [] in
```

kbd_loop runs a 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 and we have no characters saved up,
send a null character 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.

7. 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 organized 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 address 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 message sent to this address must have as data item a function of a suitable type (which is describd 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.

The underlying mechanism must now be made more complex. 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 program in Fig. 3. Address 0 is the screen, and 1 the process creator. There is now only one initial

```
screen = untag 0 messages
process_bodies = start : untag 1 messages
messages = MERGE (mapdyad fn process_bodies (2...))
fn body addr = body addr (untag addr messages)
mapdyad fn (a : as) (b : bs) = fn a b : mapdyad fn as bs
```

Fig. 3. A system supporting an arbitrary, dynamic number of processes.

user process, whose body is constructed from the function `start` and whose address is 2. More processes can be created as required.

The screen object

```
screen = untag 0 messages
screen :: [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 :: [Process_body]
Message ≡ (Address, Data_item)
Address ≡ Int (in these examples)
Data_item ≡ any type
Process_body ≡ Address → [Data_tem] → [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 expeced 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 :: [Message]
messages = MERGE (mapdyad fn process_bodies (2...))

fn : Process_body → Address → [Message]
fn body addr = body addr (untag addr messages)

mapdyad :: (* → ** → ***) → [*] → [**] → [***]
mapdyad fn (a : as) (b : bs) = fn a b : Mapdyad fn as bs
```

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 addr* 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.

8. Other possible extensions

One of the pleasing things about this model is that a number of necessary extensions appear to fit in fairly well. The features described in this section have not been implemented yet.

8.1. 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 [6]. 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 not 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.

8.2. Error recovery

This is very tricky, as there are a number of ways in which a user program could go wrong.

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 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 than 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 preventing 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.

9. 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 characterization for it has not yet been found. It may have many theoretical ramifications that have not yet been fully explored.

9.1. What are the semantics of this new system?

The formal description of programs written under this scheme appears just as hard as in a program involving nondeterministic merges, for rather than being replaced `merge` has simply been constrained. My impression, however, is that this constraint is extremely beneficial, and that its removal from the text of functional programs means that the behaviour of individual functional programs remains close to that of the lambda calculus, even if the system as a whole is able to describe more general operations.

How does the expressive power of the result compare with that of programs which allow the unconstrained use of `merge`? The model using a fixed number of processes appears to be inherently less powerful, but the general model (where process creation is allowed) can express any computation that `merge` can. The transformation from a program involving `merge` to a program using the message passing scheme is quite tricky, and corresponds quite closely to the transformation of an imperative language: our power to manipulate programs that involve `merge` is very weak.

In view of this, isn't it likely that the model with a fixed number of processes is a more desirable one? Sadly, my impression is that in writing an operating system using a functional style the ability to create new processes is extremely useful: constraining the underlying system in this way would merely force the author of the operating system to do a certain amount of extra work, without constructively aiding the structure of his program. Nevertheless, it is quite easy to be sure that process creation is not used in a program (or only used for initialization), and so a study of software written under this constraint would be an extremely interesting subject of future work.

Uncertainties concerning exactly how strict the `merge` function should be are reflected in similar uncertainty in the message passing system. The symptoms of this uncertainty appeared in the second example program (earlier in this chapter), and reference to the section on synchronization in the previous section shows that certain problems described there have not yet been eradicated.

9.2. How useful is type checking in this regime?

I have glossed over a problem concerning type checking. If the items taken as input by a process form a list then all objects sent to one particular process should be of the same 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`? Conventional 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 `Addressfor` * is a process address to which objects of type * can be sent (see Fig. 4).

```
Message ≡ (Addressfor *, *) for some *, rather than for all *
    ; e.g. if   intad :: Addressfor Int
    ;      and  chrad :: Addressfor Char
    ;      then [(intad , 3), (chrad , "a')] is a legal list
    ;                                              of messages
```

Fig. 4. Current languages seems unable to describe this type.

The type system described in [11] 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 and Miranda [12], but still unable to describe the type *Message*. Miranda has been used in examples here because it is more succinct. However, there are numerous places in the operating system where the type-checking is sidestepped. It seems a shame that this would be necessary. There are also numerous places where a new process is created and an old one discarded, simply to change the type of the input stream: it seems undesirable that the use of such a powerful facility should be necessary in order to achieve such a simple effect.

9.3. 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 an interrogate or update. How will this affect large programs if such facilities are available? It may be that programmers will be tempted to use methods just as bad as any conventional program: this remains to be seen.

Functional programming is 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.

Appendix A. An efficient implementation

The SKIM machine is a microcoded combinator reducer [18]. 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.

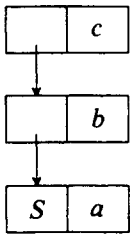
The heap

SKIM's memory is organized 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 values given in Table 1.

Table 1
Types of value on the SKIM machine.

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

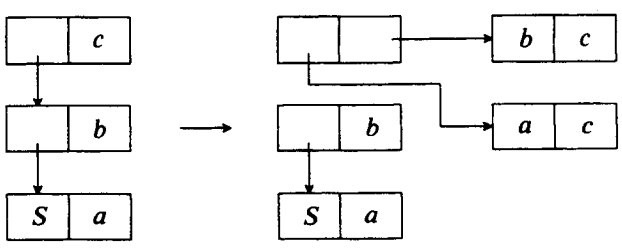
For instance, the value S a b c is represented as



The store is organized 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

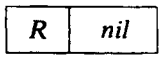
The bulk of SKIM's microcode is a normal order graph reducer. It uses pointer reversal to crawl over a graph of application pointers, which is continuously transformed until it will not reduce any furher. For instance, if given the expression *S a b c* the reducer would perform the following transformation:



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



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:

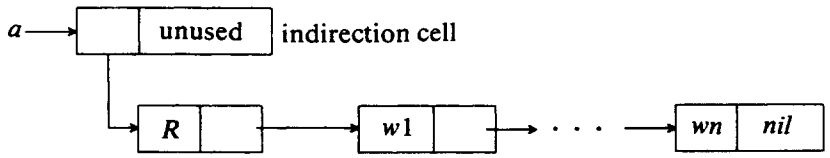
where $\rightarrow$ is an application pointer, $p\rightarrow$ is a pair pointer. (The cell on the left used to contain an R.) The I cell is necessary because the initial cell will have application pointers to it. The second cell is a pair cell, and the readin cell (where the next character will appear) has advanced to being the cell on the right.

The slicer data strutures

The microcide 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.

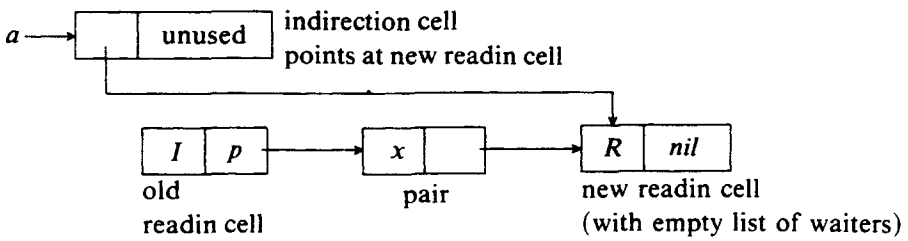
For this reason each input cell is represented by the following arrangement:



where $\rightarrow$ is an application pointer, $p\rightarrow$ is a pair pointer, $a\rightarrow$ is an address pointer.

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:



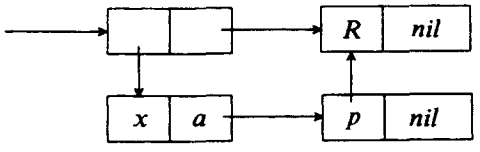
where $\rightarrow$ is an application pointer, $p\rightarrow$ is a pair pointer, $a\rightarrow$ is an address pointer.

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 toa 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 dependent on a message arriving at the readin cell. This process is removed from the active queue and added to the list in the tail part of the *R* cell. 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 further 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 evaluting 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.



where $\rightarrow$ is an application pointer, $p\rightarrow$ is a pair pointer, $a\rightarrow$ 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 which should run next if response time to devices is to be improved.

Appendix B. Miranda

Unfortunately space precludes the reproduction of a complete guide to Miranda, but some fragmentary examples of it are provided here. This will allow a reader familiar with a similar functional language to follow the examples in the text, without being confused by minor syntactic differences. More detail can be found in [20].

```
; Expressions
fn x                    ; function application, denoted by
                          juxtaposition
(fn x)                  ; brackets simply denote grouping
```

```
fn a b                  ; the same as ((fn a) b)
1 2 47 -28              ; integer constants
'a' 'x'                 ; character constants
[]                      ; the empty list
+ - * /                 ; predefined arithmetic dyadic operators
:                       ; predefined list constructor operator
[1, 2 + 4, a]           ; alternative list construction syntax
1 : (2 + 4) : a : []    ; identical expression, using colon
''hi''                  ; a string is a list of characters
++                      ; predefined append operator
(a, 'x', 23, [1,3])     ; a tuple of four lements
a + 3 WHERE a = 17      ; a WHERE clause, for naming
                          subexpressions
```

Function definitions

```
double x = x + x                    ; double is (λx. x+x)
fact x = 1, x < 2                   ; fact x is 1 if x<2,
fact x = x * fact (x - 1)           ; otherwise fact x is
                                      x * fact(x-1)
append [] y = y                     ; append x y is y if x is []
append (a:x) y = a : append x y     ; otherwise etc. etc.
```

```
; Types
; Miranda uses a static type discipline very similar to that of ML.
plus :: Int → Int → Int     ; double colon means "is of type"
                            ; arrow denotes a function type
append :: [*] → [*] → [*]   ; [x] means "list of x"
                            ; * is a type variable
Int ≡ num                   ; a type identity
Char ≡ char                 ; I like capitalized words for types
String ≡ [Char]             ; a string is a list of characters
Listof * ≡ [*]              ; declares type generator "Listof"
```

References

[1] *Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming*, Austin, TX (1984).

[2] J. Darlington, Program transformation, in: [3] 193-215.

[3] J. Darlington, P. Henderson and D.A. Turner, Eds., *Functional Programming and its Applications* (Cambridge University Press, London, 1982).

[4] D.A. Dice, Ed., *Distributed Computing Systems Programme*, IEE Digital Electronics and Computing Series **5** (Peregrinus, London, 1984).

[5] J. Fairbairn, A new type-checker for a functional language, *Sci. Comput. Programming* **6** (1986) 273-290.

[6] C.T. Haynes and D.P. Friedman, Engines build process abstractions, in: [1] 18-24.

[7] P. Henderson, Purely functional operating systems, in: [3] 177-189.

[8] P. Henderson and S.B. Jones, Shells of functional operating systems, in: [4] 290-298.
[9] P. Henderson and J.H. Morris, A lazy evaluator, in: [13] 95-193.
[10] R.J.M. Hughes, Parallel functional language use less space, Programming Research Group, Oxford University, 1984.
[11] D.B. MacQueen and R. Sethi, An ideal model for recursive polymorphic types, in: [14].
[12] R. Milner, A theory of type polymorphism in programming, *J. Comput. System Sci.* **17**(3) (1978) 348-375.
[13] *Proceedings of the Third Annual ACM Symposium on Principles of Programming Languages*, Atlanta, GA (1976).
[14] *Proceedings of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, Salt Lake City, UT (1984).
[15] W.R. Stoye, The SKIM microprogrammer's guide, Cambridge University Computer Laboratory Technical Report #40, 1983.
[16] W.R. Stoye, A.C. Norman and T.J.W. Clarke, Some practical methods for rapid combinator reduction, in: [1] 159-166.
[17] D.A. Turner, SASL language manual, University of St. Andrews Computer Science Department, 1976.
[18] D.A. Turner, A new implementation technique for applicative languages, *Software Practice and Experience* **19** (1979) 31-34.
[19] D.A. Turner, Recursion equations as a programming language, in: [3] 1-28.
[20] D.A. Turner, Miranda: a non-strict functional language with polymorphic types, *Proc. ACM Conference on Functional Programming Languages and Computer Architecture*, Nancy (1985).
[21] P. Wadler, Listlessness is better than laziness: lazy evaluation and garbage collection at compile time, in: [1] 45-52.