Abstract Machine Support for Purely Functional Operating Systems

Project Report

By

Simon B. Jones

Technical Monograph PRG-34

August 1983

Oxford University Computing Laboratory,
Programming Research Group,
8-11 Keble Road,
Oxford OX1 3QD

© 1983 Simon B. Jones

Programming Research Group,
Oxford University Computing Laboratory,
8-11 Keble Road,
OXFORD OX1 3QD
England

Address from October 1983:

Department of Computing Science,
University of Stirling,
Bridge of Allan,
STIRLING FK9 4LA
Scotland

Contents

Foreword

This document is one of a pair reporting the results of the Functional Operating Systems project commenced at Oxford in February 1982.

The report is divided into two parts: The development of an abstract machine to support a purely functional systems programming language (this document), and the exploration of a spectrum of functional, distributed operating systems (to appear later).

The two aspects of the work progressed together, driving and supporting each other. So a certain amount of the narrative text is common to both reports (in particular the Introduction), and the the reports may be read independently. Nevertheless, the reports must be taken together to provide a full record of the project, as the technical details are complementary.

I would like to acknowledge the financial support of the Distributed Computing Systems Panel of the Science and Engineering Research Council.

Thanks are due to Peter Henderson and Geraint Jones for valuable ideas and feedback during the project.

Chapter One: Introduction

Motivation:

The project is motivated by three general observations of contemporary hardware and software developments:

1) As has often been pointed out by manufacturers and researchers, the cost of computer hardware has been falling rapidly in recent years, and may continue to do so for some years yet. This has been due to improving integrated circuit technology. For example, the Hewlett Packard HP9000 series of microprocessors pack nearly half a million switching elements onto a silicon chip approximately 6mm square. Thus, not only costs but also sizes have been decreasing. These developments make it look sensible to attempt to harness the potential of many processors working in cooperation in order to construct more powerful computers. In addition, hardware experts assert that improvements in chip technology (greater density of switching elements, reduction in power consumption, etc) are approaching their forseeable limits. This lends even greater urgency to the investigation of multiple processor computer architectures as a means of achieving greater computing power.

2) In the field of programming there is increasing interest in the role of purely functional programming languages as a major weapon in the software engineer's armoury against the problem of complexity. Although the first purely functional programming language was invented in about 1960 [McCarthy], the functional style of programming has remained simply an intellectual curiosity for most of the intervening period. More recently, with growing maturity of functional programming (fp), and partly as a result of research on novel computer architectures (e.g. data flow machines [3,9], reduction machines [2,8]), fp is being more widely accepted as one direction towards advanced programming tools. In Britain ICL and GEC are both examining how fp relates to their needs for systems and applications programming.

3) One of the natural roles for fp seems to be its use in describing and implementing computer programs or systems conceived as collections of concurrently executing independent processes. (Note that there is no implication here that independent processes must be executed on independent processors.) The processes communicate via fixed channels and are thus configured as a static network determined by the channel connections. This approach leads to very clear programs in many rather sophisticated toy problems (e.g. the sieve of Eratosthenes[4]), and well modularised programs in larger, practical applications.

Taken together, these three observations suggest a rather exciting programme of research: To use some functional programming language as the systems programming language for implementing applications which are to be executed as a network of processes distributed over a network of processors. The results of such an investigation would be to extend our understanding of the potential of functional programming as a systems programming tool, to realise this potential in the form of an implementation, and to exhibit the practical value of such an approach by building useful multiprocessor systems. We would hope to demonstrate that in large practical applications the technique leads to easily managed, easily reconfigured, well modularised implementations.

Programme of research:

The starting point for the investigation had to be a small, uncomplicated implementation of a small, uncomplicated functional programming language (fpl). This simplicity was desirable since extending the language, and its implementation, would be easier, and the fundamental properties of the extensions would not be obscured. Extending a sophisticated fpl with a complex (and probably cumbersome) implementation would be neither easy nor illuminating. Thus we chose the Lispkit Lisp fpl, and its implementation as a high level abstract SECD machine[4]. Lispkit Lisp will henceforth be referred to as simply Lispkit.

Lispkit and its implementation have been modified and extended to provide a full systems programming environment when executing on a single processor. This extended system will ultimately enable a lispkit program to run interactively, to receive input from the keyboard and serial lines, to produce output on the screen and serial lines, and to interact with a disk based file store. Let us call such extended systems "functional programming computers" (fpcs).

A small collection of Lispkit fpcs will be connected via their serial line ports to give some particular network. A single Lispkit program, comprising a collection of concurrent processes, will then be distributed statically over the network to execute in a true multiprocessing fashion. A single processor in the network may support one or more processes, as may be convenient for the particular application. Communication between processes running on the same processor will occur within the machine rather than via external serial lines. The physical network of serial lines will be determined by the application, and will be reconfigured quite easily for different applications.

A typical application would be a small operating system providing a single user workstation. For example, one processor can be running an intelligent file service, another can be handling the terminal, interpreting commands and editing, and a third can be executing background jobs requested by the user. By exploiting the network of processors in this way such a system could be expected to sustain a considerable workload from the user.

Alternatively, given a collection of fpcs, a programmer could construct a stand-alone Lispkit program for some application, and could connect the fpcs in a network appropriate to that particular application. In this way the extended Lispkit fpc could provide better performance for particular applications, as well as a powerful component in a general purpose work-station.

Functional operating systems:

The progress of the project is largely driven by the requirements of the different designs of operating systems which we wish to try out. As extensions to Lispkit and its implementation become necessary, they are modified, after some deliberation, by as little as possible to maintain simplicity and cleanliness.

Many styles of operating systems may be devised within the functional framework - imagination, as usual, is the limiting factor! We have tried several distinct varieties of systems so far, but other important approaches are being investigated elsewhere [1,6].

One approach is to simply try to code a fairly conventional uniprocessing operating system (e.g. in the style of CP/M or Unix) as a single monolithic program to be run on a single fpc. This would not exploit concurrency at all. Nevertheless, experiments have shown that extremely powerful operating systems can be provided in this way.

The first step to exploiting concurrency is to devise systems comprising several stream processing functions connected in a network. An input stream is received from the keyboard (the user's commands) and a result stream is sent to the screen (the system's responses). Unfortunately the components of such systems tend to work in synchronisation, and there is no large scale concurrent activity.

The potential for large scale concurrent activity is conveniently introduced by using a stream merging (interleaving) operator [5]. The output of such a merging operator is some unpredictable (non-deterministic) mixing of the elements of the two streams. This suggests an implementation in which the producers of the streams to be merged beaver away continuously (and concurrently), presenting stream elements to the merge operator for selection.

The use of the non-deterministic choice operator in this work, and its implementation in the Lispkit machine, are quite straightforward, but the mechanism has a controversial background from the theoretical point of view[1b].

Although non-determinism (in the guise of merge) permits the the construction of systems exhibiting useful concurrency, it is by no means obvious how to exploit this potential on the user's behalf in the best way. We have started exploring designs for more sophisticated operating systems which could assist a productive user in exploiting the power available in the collection of processors at his disposal.

The Lispkit language and SECD machine architecture:

As mentioned above, Lispkit Lisp and its SECD machine implementation were chosen as the starting point for the investigation. This is a clean and simple base from which to work. The language and implementation as described in [4] provide a mechanism for executing "one shot" programs which receive all the input data, perform some computation, and produce the result, in three strictly sequential steps. The outline of a mechanism for "lazy evaluation" ("demand driven computation") is also discussed.

Thus the base language and SECD machine fall short of the requirements of the operating systems research in a number of ways:

1) A detailed mechanism for lazy evaluation is the first essential addition. The machine must be extended. The Lispkit language is not altered syntactically, but the range of programs which can be expressed in the language is considerably widened.

2) The restriction to "one shot" program execution must be removed, and a program must be allowed to work interactively between its input and output streams (typically the keyboard and screen).

3) An operator for non-deterministic choice must be introduced into the language and implementation. This involves the pseudo-parallel execution of concurrent processes on a single SECD machine.

4) Finally, in order to enable the programmer to access a range of input and output devices, the SECD machine must be extended to provide a mechanism for multiple input and multiple output streams. Most of the apparatus required is already available from the previous extensions.

The development of the extended SECD machine is closely related to similar work by Abramsky at QMC[1a].

Hardware:

Detailed arguments about the hardware to be used for running distributed systems are not a major concern of the project. However that is no excuse for not considering the matter at all.

We wish to attempt to exploit concurrency at a macroscopic level in a system. That is, substantial subsystems will be allocated statically to each processor in the network. This is in contrast to the exploitation of concurrency at a microscopic level, where there is dynamic allocation of simple tasks to processors. Examples of the latter approach are data flow machines [3,9], and reduction machines, Alice [2], ZAPP [8].

Thus we require a small collection of reasonably powerful processors (e.g. half a dozen Perqs) connected in some simple, easily reconfigured way. The distribution of parallelism at the microscopic level necessitates a large collection of small processors (e.g. 10s, 100s or 1000s of transputers) connected by a complex, general purpose communications network.

There are many groups attempting valiantly to develop and assess the latter approach in various ways, and with varying results. We have decided to opt for the former, more immediate approach.

However, beyond the intention to use a small number of powerful processors, the precise hardware techniques are not under consideration. For experimental purposes we use "off the shelf" microcomputers, such as RML 380Z, SuperBrain, Sirius, Perq and so on, as available. These machines have either one or two serial lines. We also have a custom built Mostek Z80 based computer with half a dozen serial ports which will enable more interesting networks to be constructed.

A future option could be to support all the processors and memory on a single bus. The abstraction of a collection of processors communicating via fixed channels could be provided on such hardware without the expense of bulk data transfers along serial lines. That is, perhaps, a task for someone else in the future.

Chapter Two: Lazy evaluation.

Call-by-value versus delayed evaluation:

The simplest way to execute a functional program is to adopt the call-by-value strategy used in the early chapters of [4]. The call-by-value strategy is to evaluate completely all the arguments of a function application, before proceeding to apply the function and to evaluate its body with the given arguments. In call-by-value Lispkit this extends to all primitive operations, such as arithmetic and cons, and also to let and letrec expressions, in which the local definitions are all evaluated before the main expression. This is sometimes call an "innermost" evaluation strategy, since the innermost components of an expression are evaluated before attention is turned to the expression itself.

An extremely powerful alternative is delayed, or lazy, evaluation. This is closely related to call-by-name in languages of the Algol family, in which a procedure (or function) argument is not evaluated until its value is required by the body of the procedure. (This may cause repeated evaluation of the same argument several times - resulting in confusion if any side-effects occur.) In Lispkit jargon, the argument is packaged into a "recipe", which notes the argument expression and the values of any global identifiers which it requires. Recipes are "forced" when their value is needed. In lazy evaluation an argument is not evaluated until required, but, once evaluated, the recipe is thrown away, and is replaced by the computed value. Thus no recipe will be forced more than once - avoiding repeated evaluation.

In lazy Lispkit the delayed evaluation strategy is applied to the arguments of function calls, to the arguments of each cons operation, and to the local definitions in let and letrec expressions. Delaying the arguments of cons is particularly important, as large (possibly infinite) data structures may be only partially constructed. The rest of the structure is represented by recipes, and as the structure is explored by a program the recipes are replaced by explicit structure (possibly containing embedded recipes). Thus data values are computed only as required - hence "lazy" evaluation.

Lazy evaluation is discussed at greater length in Chapter 8 of [4], where the strategy is also referred to suggestively as "call-by-need".

Stream processing functions and lazy evaluation:

A stream is simply a delayed list of s-expressions, though possibly one of unbounded length. We use the term stream to indicate that we usually think of the list as a sequence of communications from one process to another. Each process is a stream processing function - the producer of a stream has the list of messages as its result, and the consumer of the stream receives the list of messages as an argument. In the lazy evaluation strategy a stream will usually be represented, at any particular moment in the computation, by a completely evaluated initial list of elements, and a recipe describing how the stream will continue. The producer, or at least some link to the producer, will be embedded in the continuation recipe. The consumer drives the evaluation of the stream as it demands the value of each message in turn.

The lazy evaluation of potentially infinite streams is of crucial importance to our research on distributed functional operating systems.

As a simple introduction to stream processing functions consider the following definitions:

```
integersfrom(n) ≡ cons(n,integersfrom(n+1))

double(s) ≡ cons(2*head(s),double(tail(s)))

inc(s) ≡ cons(1+head(s),inc(tail(s)))
```

integersfrom(n) will generate a stream of the integers n, n+1, n+2 and so on. In particular integersfrom(0) is the stream of natural numbers.

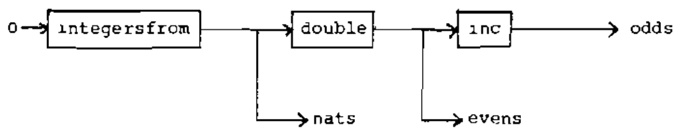
double(s) will produce a new stream whose elements are double the corresponding elements of s. In particular double(integersfrom(0)) is the stream of even numbers (starting with 0).

inc(s) will produce a new stream whose elements are one more than the corresponding elements of s. In particular inc(double(integersfrom(0))) is the stream of odd numbers (starting with 1).

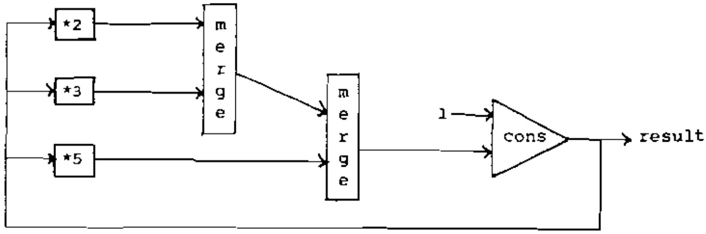
The stream definitions can be collected together

```
nats ≡ integersfrom(0)

evens ≡ double(nats)

odds ≡ inc(evens)
```

and represented pictorially as a network of channels connecting stream processing functions:



A more advanced example, taken from [4], is the generation of all numbers which are products of powers of 2, 3 and 5. The products must be generated in ascending sequence, without duplicates. The solution presented in [4] is:



where *2, *3 and *5 multiply each element of their inputs by the appropriate factor, and merge combines two ascending input streams to produce an ascending output stream with no duplicates:

```
merge(x,y) ≡ if head(x)=head(y) then merge(tail(x),y) else
             if head(x)<head(y)
             then cons(head(x),merge(tail(x),y))
             else cons(head(y),merge(x,tail(y)))
```

Extending the SECD machine for lazy evaluation:

We start from the SECD abstract machine architecture and compiler described in Chapter 6 of [4]. The notational conventions established there for abstract machine transitions and compiler rules will be retained in what follows.

First, some notes on changes of convention in the use of Lispkit keywords:

1) All Lispkit concrete syntax keywords will be written in lower case, e.g. let, letrec, lambda, etc.

2) The operation names car and cdr have been rejected in favour of head and tail in both the abstract and concrete syntaxes. They have the same respective meanings. The corresponding SECD machine instructions become HEAD and TAIL.

To implement lazy evaluation three new instructions are added to the SECD machine, and the compiler is modified in a few places. The instructions are LDE ("load expression") which constructs a recipe, AP0 ("apply to no arguments") which forces a recipe to evaluate, and UPD ("update") which overwrites a recipe with its value. The compiler changes are not extensive, and no new well formed expressions are introduced to the language.

The lazy evaluation strategy adopted here differs a little from that described in [4], but the general principle remains the same. To be precise, every well formed expression will be compiled and executed in such a way that it does not force any of its subexpressions unnecessarily, but it is certain to leave a value on the stack (i.e. an atom or cons), and not a recipe. Two advantages accrue from this: Firstly, each expression "looks after itself", and so ocurrences of AP0 are not scattered throughout the compiler. Secondly, AP0 does not need to be a "repeatedly forcing" operation.

A distinguishable structure type is added to the machine to represent recipes. This will be represented in the machine transition rules by a dotted pair enclosed in square brackets [c.e]. A recipe is rather like a closure, which is built using a cons. An important attribute of a recipe is that it may be physically overwritten by a copy of any other cell (atom or cons). This is the mechanism by which the update in place will be achieved.

The LDE machine instruction is used to delay evaluation of an expression by parcelling it up into a recipe with the current environment:

```
s  e  (LDE c.c') d   ->   ([c.e].s)  e  c'  d
```

where c is the code of the expression to be delayed (ending with UPD).

The APO instruction is used to force the top item of the stack if it happens to be a recipe. Thus there are two possible actions:

```
(x.s) e (APO.c) d   ->   (x.s) e c d    if x is not a recipe

([c'.e'].s) e (APO.c) d   ->   NIL e' c' (([c'.e'].s) e c.d)
```

The UPD instruction occurs as the last instruction in the body of a recipe. It updates the recipe with the current head of the stack (which will never be a recipe) and returns to the calling evaluation:

```
(x) e (UPD) (([c'.e'].s') e" c".d)   ->   (x.s') e" c" d
```

and the recipe [c'.e'] is overwritten with (a copy of the top cell of) x.

The compiler must be changed so that arguments to calls of user defined functions are delayed, arguments to cons are delayed, and definitions in let and letrec are delayed. Forcing operations must be inserted for expressions which might otherwise return a recipe - forcing is required after accessing a variable, and after head and tail operations.

The delaying operations:

Function application:

```
(e e1 ... ek)*n  =  (LDC NIL LDE ek*n|(UPD) CONS
                         ...    LDE e1*n|(UPD) CONS
                        e*n AP)
```

Cons:

```
(cons e1 e2)*n  = (LDE e2*n|(UPD) LDE e1*n|(UPD) CONS )
```

Let:

```
(let e (x1.e1) ... (xk.ek))*n =

                (LDC NIL LDE ek*n|(UPD) CONS
                  ...    LDE e1*n|(UPD) CONS
                 LDF e*m|(RTN)
                 AP)

                where m = ((x1 ... xk).n)
```

```
(letrec e (x1.e1) ... (xk.ek))*n =

                 (DUM LDC NIL LDE ek*m|(UPD) CONS
                      ...    LDE e1*m|(UPD) CONS

                  LDF e*m|(RTN)
                  RAP)

                 where m = ((x1 ... xk).n)
```

The forcing operations:

Variable access:

```
x*n = (LD i APO)   where i = location(x,n)
```

Head and tail:

```
(head e)*n  =  e*n|(HEAD APO)

(tail e)*n  =  e*n|(TAIL APO)
```

One more addition must be made to the compiled lazy code before it will execute successfully on the extended SECD machine. The old compiler produces code of the form:

```
( ... code to load closure for program function ... AP STOP)
```

At termination of the program, the value on top of the stack (which should be the only value on the stack) will be displayed, and therefore should not contain any recipes. Unfortunately, when the code for the program is lazy, the result on the stack may contain recipes.

To overcome this, an extra function application is inserted which explores the whole result structure, thus forcing out any recipes. The code produced then has the form:

```
( ... code to load closure for program function ... AP
  XXXX
  ... code to load closure for explore function ... AP
  STOP)
```

where XXXX is a special instruction that makes a singleton argument list:

```
(x.s) e (XXXX.c) d   ->   ((x).s) e c d
```

and the explore function, in abstract syntax is:

λ(x) . <u>if</u> finite(x) <u>then</u> x <u>else</u> UNDEFINED

<u>whererec</u> finite(x) <u>=</u> <u>if</u> atom(x) <u>then</u> T <u>else</u>
<u>if</u> finite(head(x)) <u>then</u> finite(tail(x))
<u>else</u> UNDEFINED

which itself must be compiled as lazy code (it is the APO instructions in the explore function which are important).

The need for the XXXX instruction is a slight untidiness. Its function cannot be achieved by other SECD instructions as the main arguments for the program are loaded onto the stack before any code is executed, and what we would like to do is LDC NIL before that occurs. This untidiness disappears in later extensions to the SECD machine.

This completes the extensions to the SECD machine and compiler for lazy evaluation.

Other consequences of lazy evaluation:

Various restrictions on Lispkit programs may be relaxed as a consequence of lazy evaluation. These relaxations often lead to programs with simpler structure.

The local definitions in a letrec expression may now define any type of value. Previously only function definitions were valid. In addition, mutual reference and recursion may be used in the definition of data structures. This is illustrated by the evens and odds example from earlier:

```
... whererec nats ≡ integersfrom(0)
             evens ≡ double(nats)
             odds ≡ inc(evens)
             integersfrom(n) ≡ ...
             double(s) ≡ ...
             inc(s) ≡ ...
```

Also lists may be defined by reference to themselves:

```
nats' ≡ cons(0,inc(nats'))
ones ≡ cons(1,ones)
```

As a consequence of this relaxation of letrec expressions, let expressions are effectively a redundant feature of the language.

Arguments to function applications need not have defined values, provided that the body of the function will never force a bad argument. This is not so important as its corollary, which is that local definitions in let and letrec expressions may have undefined values provided that they are never forced by evaluation of the main expression. For example, the main compiler function could be rewritten to "preselect" the fields of the various expression types:

```
comp(e,n,c) ≡ if atom(e) then ... location(identifier,n) ...
        else if rator="quote" then ...constant ...
        else ...
        else if rator="add" then ... rand1 ... rand2 ...
        else ...

        where identifier ≡ e
              rator ≡ head(e)
              constant ≡ head(tail(e))
              rand1 ≡ head(tail(e))
              rand2 ≡ head(tail(tail(e)))
              ...
```

Chapter Three: Interactive input and output

Single-shot computation versus interactive working:

Extending the SECD machine for lazy evaluation, as described in the previous chapter, does nothing to alleviate the "single-shot" nature of the computation. The compiled code expects to find a list of arguments on the stack when it starts executing. The program function is applied to these arguments. The result is explored to eliminate all recipes, and the explored structure is left on the stack to be output when the machine executes the STOP instruction. Not only is this a single-shot execution, but also the result must be a finite and acyclic structure since it must be explored before being output.

Thus the lazy programs which we can execute on such a machine may use infinite data structures as intermediate values, provided that the result is finite and that it can be computed from the initial data. A trivial example will compute a list of the first n even numbers (starting with 0), where n is the input datum:

```
λ(n) . first(n,evens)
whererec evens ≡ double(integersfrom(0))
         double(s) ≡ . . .
         integersfrom(n) ≡ . . .
         first(n,s) ≡ if n=0 then NIL
                      else cons(head(s),first(n-1,tail(s)))
```

However, it is tempting to ask for an extended implementation which will print ascending integers, starting from the input value, as requested by the following program:

```
λ(n) . integersfrom(n)
whererec integersfrom(n) ≡ . . .
```

We would expect this program to continue printing numbers for ever (possibly separated by short bursts of computation), or at least until exhaustion of memory, or maybe arithmetic overflow.

Even more exciting is the prospect of using the following program to accept a number, double it and add one, print the result, accept another number, double it and add one and print it, and so on for ever:

```
λ(kb) . inc(double(kb))
whererec inc(s) ≡ . . .
         double(s) ≡ . . .
```

The dummy identifier kb is used simply to suggest that numbers are entered from a keyboard. The numbers (or any other s-expressions) entered at the keyboard are assembled, in strict sequence, into a stream which is given to the program as its single input argument. The keyboard is only inspected for input when the program forces the delayed tail of the input stream. The result of the program is a stream, and the output driving mechanism will force and print each item of this stream in turn. Thus input and output will be interspersed, and the program will execute interactively, although remaining purely functional. The program is now a stream processing function itself.

Although the Lispkit language and its implementation arguably require other extensions in order to provide great utility, the provision of interactive input and output as outlined above immediately gives us a systems programming language of great power. For example, using no more than interactive Lispkit as described, we have implemented an s-expression structure editor which is in continual use for program development. In addition we have an interactive Lispkit interpreter, a logic language interpreter, experimental operating systems, a program source librarian, and so on.

Extending the SECD machine for interactive i/o:

The SECD machine and compiler are extended to implement the "program as a stream processing function" policy as described above. Single-shot programs will still be executable, but they must be embedded in a skeleton program which takes some fixed number of items from the input stream, applies the desired program function to them, and builds an output stream with the single result value. This brings out an important point: The input stream will always be potentially infinite (any program simply reads as much as it needs), but the output stream may be a finite list (if the program terminates it with NIL), in which case the execution of the SECD machine will terminate cleanly.

With some effort it might be possible to redesign the s-expression reading and writing routines to perform their tasks interactively, but they are outside the SECD abstract machine, and we prefer to retain simplicity in the underlying implementation. Instead the SECD machine is given a minimal interface to the s-expression readers and writers, in the form of two new instruction INPUT and OUTPUT, and the interactive i/o is handled in Lispkit itself. In fact the i/o handling is not quite pure Lispkit, since the reading and writing interface is clearly not applicative, but this interface is only used in constructing i/o drivers, and is not made available to the user through the compiler.

The only additions to the SECD machine are the two new instructions INPUT and OUTPUT. INPUT reads one s-expression from the input device and leaves it at the head of the stack. OUTPUT writes the s-expression at the head of the stack to the output device and then discards it . OUTPUT simply calls the underlying s-expression writer and so the value at the head of the stack must not contain any recipes; it must have been explored already.

The transition for INPUT is:

```
s e (INPUT.c) d   ->   (x.s) e c d
```

where x is a newly read s-expression.

The transition for OUTPUT is:

```
(x.s) e (OUTPUT.c) d   ->   s e c d
```

The STOP instruction must be changed, but this is simply residual untidiness (like XXXX, which now disappears), and is resimplified in a later chapter. The modified version of STOP is not central to the new strategy, and will be described last.

The general strategy we are now adopting is reflected in the compiled program structure:

```
(LDC NIL
     LDC NIL
     . . . code for delayed input stream expression . . .
     CONS
     . . . code for program function . . .
     AP
 CONS
 . . . code for output stream exploring and printing function . . .
 AP STOP)
```

The SECD machine is initialised by loading the code into the control register and setting the stack to NIL. No data is read during initialisation. The compiled code builds an argument list for the program function (2nd, 3rd and 4th lines above), and applies that function (5th and 6th lines). There is a single argument, which is a delayed expression containing INPUT instructions. The result of the application is built into a singleton argument list for the output driver (1st and 7th lines), which is then applied (8th and 9th lines). All output is performed by OUTPUT instructions in the third code object of the compiled program.

The special input and output code does not vary from one program function to another, and may be built into the compiler. The code may be generated from the pseudo-Lispkit given below by the main lazy compiling function described in the previous chapter, except where INPUT and OUTPUT instructions are required. The main program may be compiled in the same way - it is normal lazy code.

The input expression can be represented in pseudo-Lispkit:

```
read()
whererec read() ≡ scons(INPUT.read())
```

where INPUT stands for an occurrence of that instruction in the code, and scons ("strict cons" or "sequence cons") is like cons but the head argument is not delayed. This expression must itself be delayed (it is an argument), so it will appear as:

```
LDE ( . . . code for input expression . . . UPD)
```

When inspected, this recipe will INPUT one s-expression and make it the next item of the stream, with the tail a delayed call of the read function.

The output driving function can be represented in pseudo-Lispkit:

```
output
whererec output(s) ≡ if s=NIL then NIL else
                     if finite(head(s))
                     then OUTPUT(head(s)) ; output(tail(s))
                     else UNDEFINED
         finite(x) ≡ . . .
```

where OUTPUT(head(s)) ; output(tail(s)) indicates that the code

```
LD "s" APO HEAD APO OUTPUT
```

should be prefixed to the compiled code for output(tail(s)). Thus the semicolon indicates explicit sequencing.

The output function scans along the output stream, printing each item in turn. If the stream terminates, the function returns NIL, which will be ignored by STOP.

Unfortunately output calls itself recursively, but the SECD machine does not do tail recursion optimisation. So, as output scans further and further along the output stream it will consume more and more memory by pushing activation records onto the dump to no useful purpose. If it were not for this problem, the program which doubles, increments and prints each number entered could literally execute for ever in bounded memory.

One solution to this problem would be to modify the machine and compiler for general tail recursion optimisation. That, maybe, is a development for the future, since this is the only place in which it is necessary (and this requirement will disappear in the next chapter!). In the shorter term, the output function and the STOP instruction can be made to work together to give the required optimisation: Instead of calling itself recursively, output can return a package representing the recursive call. The package will contain the closure for output, and the argument list cons(tail(s),NIL). The activation record will have been popped from the dump when output returned the package. STOP detects the package (rather than NIL for termination) and performs the recursive call.

The pseudo-Lispkit for the output driving function is then:

```
output
whererec output(s) ≡ if s=NIL then NIL else
                     if finite(head(s))
                     then OUTPUT(head(s)) ;
                              cons(output,cons(tail(s),NIL))
                     else UNDEFINED
         finite(x) ≡ . . .
```

and the corresponding transition for STOP is:

```
(NIL.s) e (STOP.c) d   ->   Terminate cleanly

(((c'.e').args).s) e (STOP.c) d   ->   NIL (args.e') c' (s e (STOP.c).d)
```

Note that STOP is now rather like AP, which expects the stack to have the structure:

```
((c'.e') args.s)
```

Other approaches to SECD machine initialisation and constructing i/o drivers:

We are experimenting with Lispkit programs which behave as loaders of programs which are to be executed on the SECD machine. The loaders incorporate the pseudo-Lispkit input and output driving mechanisms, and may be compiled using only the main lazy compiling function. A loader is read into the SECD machine at initialisation, and expects the first item on the input stream to be a program to be executed. This program may be compiled using little more than the main lazy compiling function, since the i/o drivers are embedded in the loader.

The pseudo-Lispkit i/o drivers given on previous pages are the clearest, most concise we have devised for doing their jobs. Nevertheless, it is possible to replace some of the pseudo-Lispkit with real Lispkit, and this is done in the loader programs outlined above.

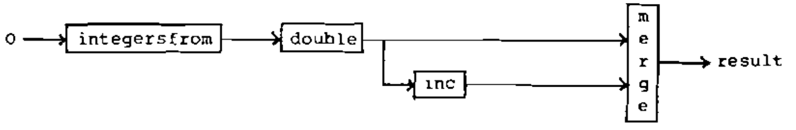
The loader program technique is proving to be an excellent way of managing the user program's i/o interface.

A collection of loaders and other utility programs has been constructed by Geraint Jones[10] to execute on the lazy interactive SECD machine.

Chapter Four: Non-determinism and pseudo-parallelism

Interleaving streams and non-determinism:

In our research on purely functional operating systems we need to express the intention that a stream is obtained by merging two or more other streams. The input sequences of elements have been arbitrarily interleaved, but the ordering of elements from each input stream is not altered. For example, if we wished, for some reason, to generate a jumbled stream of the natural numbers in which the even numbers and the odd numbers retain their own orderings, we could use the following network of stream processing functions:



This network can be represented by the following program:

```
result
whererec nats ≡ integersfrom(0)
         evens ≡ double(nats)
         odds ≡ inc(evens)
         result ≡ merge(evens,odds)
         integersfrom(n) ≡ . . .
         double(s) ≡ . . .
         inc(s) ≡ . . .
         merge(s1,s2) ≡ ? ? ?
```

in which we have no way of programming merge yet.

One possible way to implement merge is to use a simple function which alternates elements from the input streams:

```
merge(s1,s2) ≡ cons(head(s1),merge(s2,tail(s1)))
```

This certainly satisfies the criterion that the output should be some interleaving of the input streams. However, in the above example inc might be replaced by some complex function which gives a considerable delay between output elements. In the pauses it seems desirable that the stream of even numbers may continue to be processed, thus giving an unequal mixture of even and other numbers in the output stream. In our operating systems this consideration is even more important. Either input stream may be arriving from some external device, and whilst the device is inactive it is unreasonable to prevent the transmission of messages arriving on the other channel. Thus, although the solution for merge given above is adequate in some sense, it would be nice to implement merge in some more lenient fashion.

An alternative solution uses "oracle" signals to direct the merge function:

```
merge(s1,s2,oracle) ≡
          if head(oracle)=1
          then cons(head(s1),merge(tail(s1),s2,tail(oracle)))
          else
          if head(oracle)=2
          then cons(head(s2),merge(s1,tail(s2),tail(oracle)))
          else UNDEFINED
```

However, in general it is very difficult to generate the appropriate oracle messages, especially when the streams are dependent on input from external devices.

The solution to be adopted is to introduce a new expression into Lispkit which makes a non-deterministic choice between two values. The expression is:

e1 <u>or</u> e2

and may take the value of e1 or e2 arbitrarily. It is intended that the expression will be evaluated by evaluating both subexpressions e1 and e2 in parallel, and selecting whichever result is available first. The implementation of or is not allowed to ignore either e1 or e2 deliberately (for example by only evaluating e1).

Thus merge may be programmed by selecting arbitrarily between two possible result streams:

```
merge(s1,s2) ≡ alt1 or alt2
               where alt1 ≡ cons(head(s1),merge(tail(s1),s2))
                     alt2 ≡ cons(head(s2),merge(s1,tail(s2)))
```

This implementation of merge is more lenient than the alternating solution. It might ignore either input stream for ever, but that would be an unusual accident and not a design fault.

In fact there is still a technical problem with merge, which is a consequence of lazy evaluation rather than non-determinism. We would like merge to select between the alternative output streams on the basis of the "availability" of the input stream elements. However, the definition of merge given above selects between streams by the availability of the cons cells which build the alternative output streams. The components of these conses are delayed, and so there is no guarantee that either head(s1) or head(s2) is available. Thus this merge might cause deadlock by selecting an output stream whose initial element never becomes available.

The general solution to this type of problem is to apply some forcing function (e.g. finite) to the data structure component whose availability is to be guaranteed. For example:

```
merge(s1,s2) ≡ alt1 or alt2
               where alt1 ≡ if finite(head(s1))
                            then cons(head(s1),merge(tail(s1),s2))
                            else UNDEFINED
                     alt2 ≡ if finite(head(s2))
                            then cons(head(s2),merge(s1,tail(s2)))
                            else UNDEFINED
```

In particular cases the forcing function may be simpler. For example, if the availability of "something" rather than "everything" is required:

```
merge(s1,s2) ≡ alt1 or alt2
               where alt1 ≡ if here(head(s1))
                            then cons(head(s1), . . . )
                            else UNDEFINED
                     alt2 ≡ if here(head(s2))
                            then cons(head(s2), . . . )
                            else UNDEFINED
here(x) ≡ if atom(x) then T else T
```

Extending the SECD machine for non-determinism:

As described above, non-determinism is to be introduced into Lispkit through the or operator. This clearly requires the addition of a new instruction, OR, to the SECD machine. However, the alterations to the abstract machine must be far more extensive as the non-deterministic choice requires that the alternative expressions be evaluated in parallel. The strategy to be implemented is that all evaluations of recipes will occur as parallel processes which "share time" on a single SECD machine. The new abstract machine will be a pseudo-parallel SECD machine. Each APO instruction will initiate a new process if it needs to force a recipe. Each UPD instruction will terminate a process. Each OR will simultaneously force two recipes, one for each alternative subexpression.

The modified SECD machine is potentially far more powerful, as eventually pseudo-parallelism could be replaced by true parallelism (for example, on a multiprocessor machine such as Alice[2]). The the mechanism could be extended, quite naturally, to evaluate the subexpressions of arithmetic operators simultaneously, and so on. We shall not pursue this line of development here.

First we must develop a new, process oriented strategy for lazy evaluation, and then the non-deterministic choice mechanism will be a small further step.

The abstract machine needs a new, distinguishable structure type to represent a process. When a recipe is forced, and its evaluation becomes a parallel process, the recipe is altered to be a process. A process cell has no subfields; it is simply a placeholder for the value of the recipe. This value will eventually be installed by an UPD instruction. A process cell will be represented by a pair of curly brackets {}. The new type is necessary in order to identify recipes which are already evaluating, so that the recipe is not forced a second (or further) time by APO instructions in other parallel processes.

Since we are now dealing with a multiprogramming abstract machine there must be apparatus for process scheduling:

The process which is executing is held in the machine registers S, E, C and D. Processes which are idle are kept in one of two new registers READY and DONE. Processes in READY have not yet received a time slice in the current round of scheduling. A process is executed by transferring it from READY to S, E, C and D, and at the end of its time slice to DONE. When READY is empty, the contents of DONE is transferred to READY, and DONE is cleared. Time slices are terminated by either an UPD instruction (when the process dies), or by an APO instruction which does not find a value on the stack. Thus processes voluntarily relinquish the CPU. This mechanism could easily be replaced by instruction counting to enforce fair time slicing, but the former method has a lower overhead per instruction executed, and in lazy evaluation APO and UPD instructions are executed quite frequently.

Since READY and DONE are built as s-expression stacks the scheduling mechanism is rather unusual, but very simple and adequately fair. An important consideration is that new processes are added to DONE and not to READY, so that the reproductive descendants of a reproductive parent process do not prevent other processes from progressing.

There is no special treatment required for processes which are waiting, as all processes wait busily. Busy waiting occurs when APO forces a recipe and must wait for the process cell (the recipe) to receive its value. To have APOs waiting busily in this way sounds rather extravagant: Nested forcings will give several busily waiting processes for a single usefully active process (at the end of the chain), and in a pseudo-parallel system several APOs may be waiting busily for the same process to terminate. However, in an experiment which kept a queue of waiting processes in a subfield of each process cell, execution speed increased by only about 10 per cent. The former method was adopted because it is simpler, and also because the implementation of OR cannot make use of the optimisation, and it is better to have one mechanism for the job than two.

In order to describe the new transitions for APO and UPD (and later OR), and at the same time the process swapping operation, we shall add READY and DONE to the SECD quadruple, and also make use of a special instruction DISPATCH. DISPATCH does not appear in the SECD implementation, although there is no reason why it should not; here it is simply a descriptive device. When the next process is to be executed the DISPATCH instruction is installed in the control register. Transitions will be given for DISPATCH as if it were an abstract machine instruction; these transitions describe the scheduling mechanism.

The new transition for APO must handle three cases: When the value is ready, when a recipe must be forced, and when a process is still evaluating: (Note: A hyphen in place of S,E or D means that the actual contents are unimportant)

```
(x.s) e (APO.c) d ready done -> (x.s) e c ready done
          where x is not a recipe or process

(x.s) e (APO.c) d ready done ->

                - - (DISPATCH) - ready ( NIL e' c' x            *1
                                          (x.s) e (APO.c) d     *2
                                          .done )
                where x is a recipe [c'.e'],
                      x is altered to be a process cell,
                      *1 is the new process,
                  and *2 is the suspended current process

(x.s) e (APO.c) d ready done ->

                - - (DISPATCH) - ready ( (x.s) e (APO.c) d      *1
                                          .done )
                where x is a process {},
                  and *1 is the suspended current process
```

The transition for UPD is still quite simple:

```
(x) e (UPD) d ready done -> - - (DISPATCH) - ready done
                where d will be a process {} which is overwritten
                      by (a copy of the top cell of) x
```

Note that the initial dump of a newly created process is the recipe/process which is eventually overwritten by UPD.

The transition rules for DISPATCH are also simple:

```
- - (DISPATCH) - NIL NIL -> Halt the machine

- - (DISPATCH) - NIL done -> - - (DISPATCH) - done NIL
                where done is not NIL

- - (DISPATCH) - (s e c d.ready) done -> s e c d ready done
```

It is now easy to implement the non-deterministic choice operator using the above apparatus. The following rule is added to the compiler:

```
(or e1 e2)*n = (LDE e1*n|(UPD) LDE e2*n|(UPD) OR)
```

and the OR instruction is added to the SECD machine with the following transitions:

```
(x y.s) e (OR.c) d ready done -> (z.s) e c ready done
     where either x or y is a value (neither recipe nor process),
       and z is that value (x or y as appropriate)

(x y.s) e (OR.c) d ready done ->

                    - - (DISPATCH) - ready ( "xprocess"
                                             "yprocess"
                                             (x y.s) e (OR.c) d
                                             .done )
     where neither x nor y is a value,
       and "xprocess" and "yprocess" are present if the
         corresponding x or y is a recipe (which must be forced),
         and absent if it is a process. If x is a recipe [c'.e']
         then "xprocess" is the new register set
                   NIL e' c' x
         and x is altered to be a process.
         Similarly for y and "yprocess".
```

Some words of explanation are appropriate. To make the non-deterministic choice e1 <u>or</u> e2, e1 and e2 are submitted as two new processes by OR. The process which executes OR then has two processes at the head of its stack, and waits busily, re-executing OR, until one of the two processes on the stack is found to have been updated to a value. That value is then retained on the stack, the other (probably still a process) is discarded, and the choice has been made on the basis of availability.

Although discarded, the process computing the rejected alternative is still known to the scheduling mechanism, and so will continue executing. It is well known that it is extremely difficult to kill the unwanted process - it may itself have started new processes, some of which may be forcing globally known recipes and must either be allowed to terminate or be reset carefully to their unforced state. Fortunately, when executing lazily it is reasonably economic, though not perfectly so, to leave the processes executing. As a consequence of lazy evaluation the process will terminate "fairly soon", usually having computed an atomic or partially constructed result. The discarded process cell (still, and only, known to the evaluating process) will be updated and the process will kill itself. Any globally known recipes which are incidentally forced by the process will appear to other processes to be properly updated values. Thus in a purely functional system the side effects of concurrent processes are entirely benevolent, which is not true of the potentially chaotic behaviour of programs in traditional languages endowed with parallel tasking "facilities".

The non-deterministic, pseudo-parallel SECD abstract machine is entirely compatible with code produced by the compiler from the previous chapter. Only the rule for compiling or must be added.

Rewriting the output driving function:

With the new SECD machine described above it is possible to solve the tail recursion optimisation problem in the output driving function in a different way. In the new scheme no "application package" needs to be constructed, and the STOP instruction simply terminates the current process.

Here is the new output driver, in pseudo-Lispkit:

```
output
whererec output(s) ≡ if s=NIL then NIL else
                     if finite(head(s)) then
                          OUTPUT(head(s)) ; (NIL or output(tail(s)))
                     else UNDEFINED

          finite(x) ≡ . . .
```

and the corresponding transition for STOP is:

```
s e (STOP) d ready done -> - - (DISPATCH) - ready done
```

The expression (NIL or output(tail(s))) is the crucial feature of this output driver. The expression returns NIL immediately and the current call of output returns it, thus popping the activation record from the dump. Meanwhile, the discarded recursive call continues independently. It will print an item, and then return NIL to update the discarded process cell and die. But it will have created another independent recursive call, and so on.

The scheme is still not entirely satisfying, as it relies on two properties of the implementation of or. Firstly, that OR does not kill the discarded process, and secondly that the dump of the process executing OR is not donated to the child processes. In this respect OR is being used to simulate an explicit parallel process generator e1 par e2, which returns the value of e1, but incidentally starts a new process for e2 and then forgets it without killing it. A PAR instruction could eventually be added to the machine to give explicit existence to this tool for constructing output drivers. The compiler rule and machine transition would be:

```
(par e1 e2)*n = (LDE e2*n|(UPD) PAR)|e1*n

([c'.e'].s) e (PAR.c) d ready done ->
                              s e c d ready (NIL e' c' {}.done)
```

Chapter Five: Multi-stream input and output

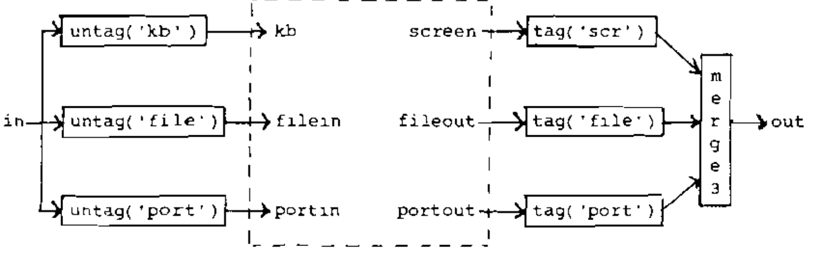
Extending interactive i/o to other devices:

The interactive SECD machine developed in the preceding chapters is able to execute programs which receive a single input stream and generate a single output stream. Usually these streams are from the keyboard, and to the screen, respectively, but we have used devious means at a very low level in the implementation to switch these streams to and from disk files. In this way it is possible, for example, to use the Lispkit s-expression editor to modify Lispkit programs kept in disk files.

However it is clearly desirable, for general systems programming, to enable a Lispkit program to control its own input and output, to and from the terminal and file store, explicitly and cleanly. In addition our research on distributed operating systems demands that Lispkit programs should be able to perform input and output of s-expressions via the hardware serial ports.

Two quite simple solutions present themselves:

Firstly, we could retain the single i/o stream interface between a Lispkit program and the i/o drivers, but tag each arriving s-expression with some identification of its origin, and each departing s-expression with some identification of its intended destination (the latter would be the responsibility of the Lispkit program). A typical program to execute on such a system would have the following network of stream processing functions:



in will be a stream of items from the keyboard, file store, and serial port tagged (by the input driver) with 'kb', 'scr' and 'port' respectively. The dotted box contains some application program network computing the output streams from the input streams. merge3 is a three way non-deterministic merge, built quite easily from two way merges. The tagged stream out will be decoded by the output driver and low level s-expression output software. untag generates a stream processing function which filters and removes tags from its input stream. tag generates a stream processing function which tags each item of its input stream. The overall program could have the following structure:

```
λ(in). { merge3(tag('scr')(screen),
                tag('file')(fileout),
                tag('port')(portout))

         whererec kb ≡ untag('kb')(in)
                  filein ≡ untag('file')(in)
                  portin ≡ untag('port')(in)
                  screen ≡ f(kb,filein,portin)
                  fileout ≡ g(kb,filein,portin)
                  portout ≡ h(kb,filein,portin) }

whererec merge3(s1,s2,s3) ≡ . . .
         untag(id)(s) ≡ if head(head(s))=id
                        then cons(tail(head(s)),untag(id)(tail(s)))
                        else untag(id)(tail(s))
         tag(id)(s) ≡ cons(cons(id,head(s)),tag(id)(tail(s)))
         f(s1,s2,s3) ≡ . . .
         g(s1,s2,s3) ≡ . . .
         h(s1,s2,s3) ≡ . . .
```

The alternative solution is to absorb the untagging, tagging and merging operations into the i/o drivers (and thereby possibly not do them at all). The program would then correspond roughly to the dotted box in the diagram above. A simple interface between the i/o drivers is for the input driver to supply the program with a single argument which is a short list of streams, one from each input device, and for the program to produce a list of streams to be decoded by the output driver. The position of the stream in the list will determine the i/o device used - there will be no tagging Thus on a machine with a terminal, a file store and one serial line a typical program could have the structure:

```
λ(in). { cons(screen,cons(fileout,cons(portout,NIL)))

         whererec kb ≡ head(in)
                  filein ≡ head(tail(in))
                  portin ≡ head(tail(tail(in)))
                  screen ≡ f(kb,filein,portin)
                  fileout ≡ g(kb,filein,portin)
                  portout ≡ h(kb,filein,portin) }

whererec f(s1,s2,s3) ≡ . . .
         g(s1,s2,s3) ≡ . . .
         h(s1,s2,s3) ≡ . . .
```

The latter scheme has been implemented. It is rather simpler since, in the former scheme, the messages directed to each device must be separated from each other at some level in the output system (either in the pseudo-Lispkit output driver or in the underlying s-expression output routines), and so the effect of the merging is undone. In the latter scheme there is no merging and no unmerging.

The next matter to be decided is the nature of the communications along each i/o stream. Debate on the precise properties of this interface is continuing, but the following simple scheme has been implemented to test the feasibility and utility of some form of multi-stream i/o. The adopted scheme is sufficiently powerful to permit an interesting range of experiments on distributed operating systems.

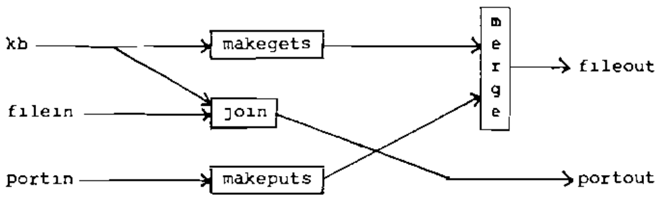
Input from the keyboard and output to the screen remain as they have been previously in the interactive SECD machine. S-expressions entered at the keyboard arrive as the input stream, and the s-expressions of the result stream are displayed on the screen.

Input and output via the serial ports is treated in the same way as i/o via the terminal - s-expressions are sent and received. Each serial port is associated with one input and one output stream.

However, the file store is, by necessity, rather different. Each file will contain exactly one s-expression. Clearly then, items in the file store output stream which are to be written to files must carry a file name with them to identify their destination on the backing store. But the output stream must also contain requests for files which are to be input - the contents of those files will be the items appearing on the file store input stream. Thus the output stream consists of commands to the file store, the most important of which will be "(get filename)" and "(put filename filecontents)". The former will cause the contents of the named file to be added to the input stream. The latter will create (or overwrite) the named file with the given contents, with no response appearing on the input stream. Clearly the little command language could be extended with delete, rename, directory request, and so on. It is important that the file store actions are carried out in precisely the order in which they appear in the output stream. A convenient format for file names is to allow them to be either an atom or a consed pair of atoms (in which case the underlying software can form a single name suitable for the given external file store).

Note that this interface to the file store is very similar to the interface to the simple databases described in [5]. There Henderson shows how a file store can be implemented in a purely functional way, and so we have not brought something essentially non-applicative into Lispkit by the use of such a store - although it will usually be implemented in a non-applicative way by overwriting areas of disk.

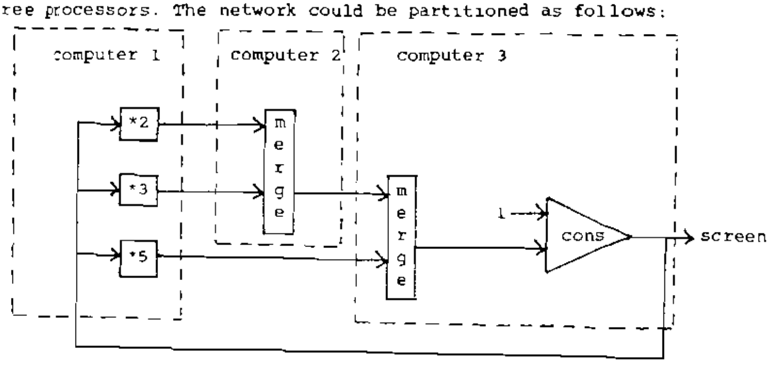
The example program below uses this interface to send files named at the keyboard out along the serial line, and to enter files arriving along the serial line into the file store. Each message passing along the serial line is a 2-list "(filename contents)". This could be used as the basis for a more sophisticated machine to machine file transfer system:



```
λ(in). { cons(NIL,cons(fileout,cons(portout,NIL)))

        whererec kb ≡ head(in)
                 filein ≡ head(tail(in))
                 portin ≡ head(tail(tail(in)))
                 fileout ≡ merge(makegets(kb),makeputs(portin))
                 portout ≡ join(kb,filein)  }

whererec makegets(s) ≡ map(λ(x).cons('get',cons(x,NIL)),s)
         makeputs(s) ≡ map(λ(x).cons('put',x),s)
         join(s1,s2) ≡ cons(cons(head(s1),cons(head(s2),NIL)),
                            join(tail(s1),tail(s2)))
         map(f,s) ≡ . . .
         merge(s1,s2) ≡ . . .
```

Given a collection of computers each supporting a multi-stream Lispkit system, any program previously conceived as network of communicating processes may now be physically distributed. This is achieved simply by partitioning the network into groups of stream processing functions (preferably connected groups), and assigning the communication channels connecting the groups to hardware serial lines. The single Lispkit program describing the original network is similarly transformed by naming each of the channels which are to correspond to serial lines, partitioning the statements into the appropriate subnetworks, and writing down each subnetwork as a separate program. The separate programs are executed on the collection of computers which have been connected by serial lines corresponding to the group connections required.

For example, the network which solves the powers of 2, 3 and 5 problem, discussed in Chapter Two, can easily be distributed over a group of, say, three processors. The network could be partitioned as follows:



Computer 1 will use one serial line for input, and three serial lines for output; it needs three serial lines altogether, since the input from and output to computer 3 can share the same line. Computer 2 will use two serial lines for input, and one for output; it needs three lines altogether. Computer 3 will use two serial lines for input, one line for output, and also generates a stream for the screen; it needs two serial lines altogether, since the two channels to computer 1 can share the same line. The program to be executed on computer 1 would look like this:

```
λ(in). { cons(NIL,cons(NIL,cons(port1out,
                               cons(port2out,cons(port3out,NIL)))))

          whererec port3in ≡ head(tail(tail(tail(tail(in)))))
                   port1out ≡ times2(port3in)
                   port2out ≡ times3(port3in)
                   port3out ≡ times5(port3in) }

whererec times2(s) ≡ . . .
         times3(s) ≡ . . .
         times5(s) ≡ . . .
```

where serial port 3 has been used as the channel to computer 3.

Extensions to the SECD machine and compiler:

The abstract machine itself needs to be changed very little to enable multi-stream i/o as described above - most of the required apparatus has already been provided. The greatest changes occur in the input and output drivers supplied by the compiler (or loader system), and in the low level s-expression i/o routines which must now handle each device according to its needs.

In the modified machine the INPUT and OUTPUT instructions will expect to find a numeric atom on top of the stack which identifies the device to be used - the convention adopted is: 0=terminal, 1=file store, 2,3,etc are serial ports. In addition to this, each INPUT and OUTPUT operation must wait busily if the required device is not yet ready to engage in the communication; this is to ensure that other processes in the machine may continue to execute while the particular input is not available. For example, the screen is always ready to accept output, but the keyboard is not considered to be ready until the user has typed, say, one useful character or maybe a complete line of text. The properties of the other devices in this context will be discussed later.

The transition for INPUT and OUTPUT are thus:

```
(n.s) e (INPUT.c) d ready done  ->
              - - (DISPATCH) - ready ( (n.s) e (INPUT.c) d.done )
         if device n is not ready for input

(n.s) e (INPUT.c) d  ->  (x.s) e c d
         if device n is ready for input,
         and the next s-expression is x
```

```
(n x.s) e (OUTPUT.c) d ready done  ->
          - - (DISPATCH) - ready ( (n x.s) e (OUTPUT.c) d.done )
          if device n is not ready for output

(n x.s) e (OUTPUT.c) d  ->  s e c d
          if device n is ready, x is output
```

The input expression which is supplied to a program must evaluate to a list of delayed stream input expressions of the kind used in the previous chapter. It is quite simple:

```
input(0)
whererec input(n) ≡ cons(instream(n),input(n+1))
         instream(i) ≡ scons(INPUT(i),instream(i))
```

where scons, as before, does not delay its first argument, and INPUT(i) compiles as follows:

```
"INPUT(i)"*n  =  i*n|(INPUT)
```

Note that this input driver will only attempt to read from devices whose streams are actually accessed by the program.

The output driver must generate a process to follow each output stream from the result of the program. The result of the program will be a short list of streams, and so there will be a small number number of such processes. Each process will force its own stream independently, and will disappear from the machine if it encounters the end of a finite stream. We can use the same trick to generate the separate processes as we do to scan and print each stream:

```
λ(out). output(0,out)
whererec output(n,l) ≡ if l=NIL then NIL
                       else (outstream(n,head(l))
                             or output(n+1,tail(l)))
         outstream(n,s) ≡ if s=NIL then NIL else
                          if finite(head(s))
                          then OUTPUT(n,head(s)) ;
                               (NIL or outstream(n,tail(s)))
                          else UNDEFINED
         finite(x) ≡ . . .
```

where OUTPUT(n,tail(s)) compiles:

```
"OUTPUT(n,tail(s))"*m = s*m|(TAIL AP0)|n*m|(OUTPUT)
```

Note that both of these drivers have been constructed to work correctly on any hardware - they are independent of the presence of any particular devices. Hence the same compiler can be used for any machine. The Lispkit program must of course be consistent with the machine on which it is executing - it must only attempt to communicate with devices recognised by the particular implementation.

Apart from the input and output drivers there is no other change to the compiler.

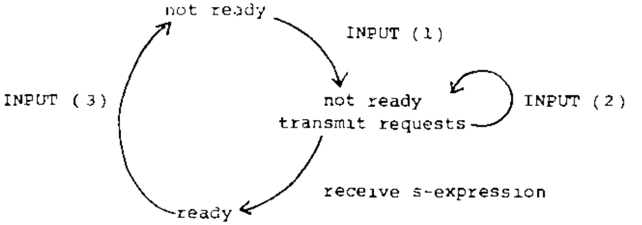
Lower level device control:

It only remains to discuss a useful scheme for handling the various i/o devices below the level of the SECD abstract machine. In practice this means deciding when to perform s-expression input and output, and when the devices are ready or not ready for the transaction. Guided by the general principle of laziness, we will attempt to ensure that no s-expressions are input until they have been requested by an INPUT instruction, and that no OUTPUT instruction may proceed until the s-expression which it provides has been accepted by the output device. This means that on each output stream the driver is always preparing the next item for output; this does not quite conform to the laziness we might expect, in which an OUTPUT instruction is not allowed to proceed to prepare the next item until the device has become ready for output, but it is a useful strategy.

As mentioned above, the screen is always ready for output and the output item will be displayed. The keyboard will be ready for input when some useful quantity of text has been typed (for example, a complete line of text containing at least the start of an s-expression). Once an s-expression has started, attention is devoted to the keyboard until the expression is complete. This is a simple scheme which enables the keyboard to be inspected only on demand from the program.

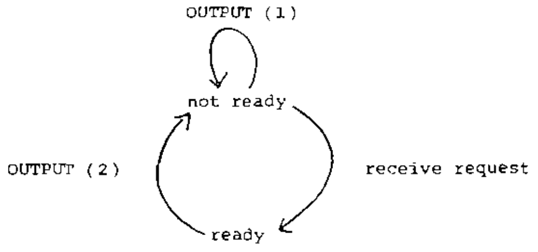
The serial lines are a little more complicated. To maintain the demand drive policy, and to economise on buffer space (in the list cell heap), we would like to delay transferring an s-expression from the producer's machine to the consumer's machine until the consuming program has requested the next stream item by executing an INPUT instruction. This effect is achieved if an INPUT instruction causes a control signal to be transmitted along the serial line requesting an s-expression to be sent by the producer. INPUT then waits busily until an s-expression has been received, i.e. until the serial port's receive buffer becomes ready. Conversely, an OUTPUT instruction for a serial line must wait busily until a request has been received from the consumer. This is the outline of a demand driven s-expression transfer protocol which could probably be implemented in several ways. Of course, this protocol will be built on a lower level, reliable, full duplex protocol of some kind (which allows the same transfer strategy to be used in both directions). At the lower level the serial line could be driven either by interrupts or, for example, by regular polling between each SECD instruction or at each process swap.

The serial line input controller cycles through three states:



where INPUT (1), (2) and (3) are re-executions of the same INPUT instruction. (1) starts transmission of request signals. (2) is the busy waiting phase. (3) finally accepts the s-expression which has been received.

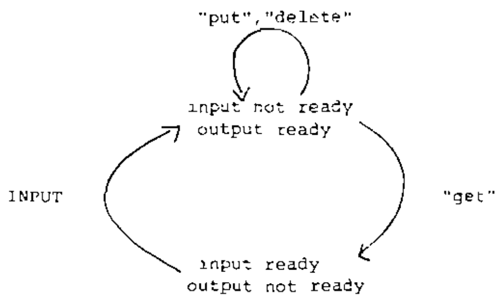
The serial line output controller has only two states:



where OUTPUT (1) is the busy waiting phase and OUTPUT (2) is the same instruction, and provides the requested output.

The file store is slightly different again, since the input and output streams are coupled. We must satisfy two constraints here. Firstly that the actions appearing in the file store output stream are performed strictly in sequence, and secondly that files requested for input by a "get" command are not read from the file store until the next item on the file store input stream is demanded by the program. The following strategy satisfies these requirements: The file store is initially not ready for input, and ready for output. When ready for output a "put" command creates (or overwrites) the named file, and the process proceeds. Similarly a "delete" or "rename" could occur immediately, and leave the file store ready for output. A "get" command will allow the output process to continue, but the file store becomes not ready for output, ready for input, and the file name is noted. When ready for input an INPUT instruction receives the contents of the file whose name has been noted, and causes the file store to become not ready for input, and ready for output again.

The file store passes round a small cycle:



where "put", "delete", and "get" are specific instances of commands output by OUTPUT.

Chapter Six: Summary of (some of the) residual problems with the SECD machine

The alterations to the SECD machine that have been described in the previous chapters of this report are important for several reasons: With only modest, and reasonably easily understood, changes to the abstract machine the power of the machine to support general programming has been increased considerably. This establishes a direction in which the machine itself could be further improved without substantially altering the programming interface to the system. The new abstract machine is sufficiently powerful to test out many interesting ideas concerning the use of purely functional programming for systems programming - ideas which are essentially to do with the language and programming style rather than any particular implementation.

However, there are aspects of the abstract machine and its use which leave something to be desired, although the consequences are only dire for rather pathological programs. Seven problems are listed below. The first is a problem with the simple implementation of laziness. The second is an inevitable consequence of the universal application of the lazy evaluation strategy (in whichever way the laziness is actually implemented). The third and fourth problems concern the rather simple implementation of non-determinism. The fifth and sixth problems are not faults with the implementation, but rather places where a re-design might yield a better, or more general, systems programming environment. The seventh problem identifies an inefficiency which lends itself to a solution in special purpose hardware.

1) The instructions which build function closures and recipes, AP0 and LDE, bind the entire current environment into the new object. Thus nothing in the environment may be collected as garbage until the closure or recipe itself is collectable (for example potentially lengthy input or output streams). It would be attractive, though for small programs possibly less efficient, to bind into closures and recipes only those variables currently in scope which may be referenced by the body of the closure or recipe (i.e. the free variables of the expression).

2) The need to use little sequence enforcing constructs, as in merge and the input and output drivers, is rather untidy. Interactive Lispkit programs must occasionally resort to such constructs to ensure that lists of queries sent to the screen and responses read from the keyboard are interleaved correctly - for example, the next output can be delayed by making it depend on an application of finite to the previous input.

3) The non-deterministic instruction OR does not terminate the process (or its descendants) which is computing the discarded alternative. In many cases this will not matter, it will simply lead to temporary inefficiency as the processes continue to use the machine before terminating themselves. However, in the case of an extremely expensive, or even non-terminating, discarded alternative the consequences could be disastrous.

4) The stream merging function which can be implemented using OR provides no guarantee of fairness - it might accidentally ignore one input stream indefinitely. The solution to this would probably involve replacing the OR instruction with a stream merging instruction as the primitive source of non-determinism.

5) There would probably be advantages in handling input and output streams as sequences of single characters rather than sequences of s-expressions. This would open up the possibility of processing general text, and controlling devices in more detail. The program, or maybe the input and output drivers, would then be responsible for parsing input text, and formatting output text.

6) The differences between the terminal, serial line and file store interfaces to Lispkit programs could be simplified and made more uniform by treating the keyboard, screen and serial line ports as special files with distinguishable names. For example, in order to obtain the stream of inputs from the keyboard a program might output the request "(get kb:)".

7) There are two inadequacies in the use of serial lines for interprocessor communication: Firstly, the transmission of s-expressions between processors via serial lines is tediously slow, and unfortunately timesharing on the SECD machine comes to a halt while such transmission is occurring. Less pedestrian low level protocols, or use of parallel lines, would increase speed - but not appreciably if the s-expression syntax routines used for input and output are the limiting factor. (Adoption of character i/o streams, as in 5) above, would enable timesharing to continue during s-expression transmission.) Secondly, only acyclic, recipe free structures may be transmitted via the serial lines, since they are transmitted in external s-expression syntax (note that this essentially rules out transmission of closures, which are usually recursive). A possible solution to both of these problems is to design a special purpose computer with several processors on a single bus and accessing a common large list store. Thus transmission of any Lispkit value or recipe is then possible simply by exchanging a pointer to the item.

These problem areas, and others, will be considered during the continuing development of Lispkit and the SECD machine as systems programming tools.

References

[1a] S.Abramsky: SECD-M: A virtual machine for applicative multiprogramming. Queen Mary College, Computer Systems Laboratory, 1982.

[1b] S.Abramsky: A simple proof theory for non-deterministic recursive programs. Queen Mary College, Computer Systems Laboratory, 1982.

[2] J.Darlington and M.Reeve: Alice, a multiprocessor reduction machine for the parallel evaluation of applicative languages. Internal report, Dept of Computing, Imperial College, 1981.

[3] J.B.Dennis: Varieties of data flow computers. Proc. of 1st Int. Conf. on Distributed Computer Systems, pp430-439, October 1979.

[4] P.Henderson: Functional programming: Application and implementation. Prentice-Hall, London, 1980.

[5] P.Henderson: Purely functional operating systems. In Functional programming and its applications, Eds Darlington, Henderson and Turner, CUP 1982.

[6] K.Karlsson: Nebula: A functional operating system. Internal report, Laboratory for Programming Methodology, Chalmers University of Technology and University of Goteborg, 1981.

[7] J.McCarthy et al: The Lisp 1.5 Programmer's Manual. MIT Press, 1962.

[8] F.W.Burton and M.R.Sleep: Executing functional programs on a virtual tree of processors. Proc. ACM Conf. on Functional Programming Languages and Computer Architecture, October 1981.

[9] I.Watson and J.Gurd: A prototype dataflow computer with token labelling. Proc. Nat. Comp. Conf. Vol 48, pp623-628, 1979.

[10] P.Henderson, G.A.Jones, S.B.Jones: The Lispkit Manual. Programming Research Group Technical Monograph PRG-32, Oxford University, 1983.

Appendix: Multi-stream machine input and output revisited

This note is a short account of alterations to the behaviour of the INPUT and OUTPUT instructions, and to the output driver given in Chapter 5. The alterations solve two problems associated with mechanisms described in that chapter. Firstly, it was not possible to describe the INPUT and OUTPUT transitions independently of the details of device readiness and device control. Secondly, the output driver did not prepare output values only when required, but in advance, in anticipation of the requirement for output. This was satisfactory when driving a terminal (where the screen always becomes ready eventually), but, for example, the serial line may never request the next output.

Solving the first problem essentially means tidying up the abstract SECD machine and its description. Solving the second problem will ensure that the demand propagation strategy between machines is correctly implemented.

New mechanisms:

Each device is given a collection of buffers and flags. For input the device has an s-expression buffer register IBUF, and two flags IREQ and IBUFRDY. For output the device has an s-expression buffer register OBUF, and two flags OREQ and OBUFRDY. The SECD machine register set now includes short vectors of IBUF, IREQ, IBUFRDY, OBUF, OREQ and OBUFRDY registers - one element of each vector per device. These are the only interface between the abstract SECD machine and the devices. Low level software, which need not be considered in detail here, is responsible for performing device control in accordance with the register vectors and device statuses. This could be done in a (sufficiently) frequently activated polling routine, or a concurrently executing process.

For input (for each device n):

Flags IREQ(n) and IBUFRDY(n) are initially false.

An INPUT instruction for device n sets IREQ(n) true to request input, and then waits busily until both IREQ(n) and IBUFRDY(n) are true. INPUT then loads IBUF(n) onto the stack, clears IREQ(n) and IBUFRDY(n), and continues program execution.

Meanwhile, the polling routine does nothing with device n until IREQ(n) is true, IBUFRDY(n) is false, and device n has input available. It then reads an s-expression from the device, deposits it in IBUF(n) and sets IBUFRDY(n).

Transitions for INPUT:

(n.s) e (INPUT.c) d ready done $\neg$IREQ(n) $\neg$IBUFRDY(n) ->

- - (DISPATCH) - ready ((n.s) e (INPUT.c) d.done)
IREQ(n) $\neg$IBUFRDY(n)

(n.s) e (INPUT.c) d ready done IREQ(n) $\neg$IBUFRDY(n) ->

- - (DISPATCH) - ready ((n.s) e (INPUT.c) d.done)
IREQ(n) $\neg$IBUFRDY(n)

(n.s) e (INPUT.c) d IREQ(n) IBUFRDY(n) IBUF(n)=x ->

(x.s) e c d $\neg$ IREQ(n) $\neg$IBUFRDY(n)

Cycle for polling routine:

Start

↓

Wait until IREQ(n) and IBUFRDY(n) and device n
has input available

↓

Read s-expression into IBUF(n) and set IBUFRDY(n)

For output (for each device n):

The OUTPUT instruction expects to find on the stack a device number and a recipe, process or fully evaluated s-expression representing the next item to be output.

Flags OREQ(n) and OBUFRDY(n) are initially false.

An OUTPUT instruction for device n waits busily until OREQ(n) is true and OBUFRDY(n) is false. It then forces a recipe or waits for a process to complete if necessary. When the s-expression is fully evaluated it is loaded into OBUF(n), OBUFRDY(n) is set and program execution continues.

Meanwhile, the polling routine waits until OREQ(n) is false and device n is requesting (or otherwise needing) output. OREQ(n) is set and the routine waits until both OREQ(n) and OBUFRDY(n) are true. The contents of OBUF(n) are sent to device n, and both OREQ(n) and OBUFRDY(n) are cleared.

Transitions for OUTPUT (compare APO):

```
(n x.s) e (OUTPUT.c) d ready done   ¬OREQ(n) ¬OBUFRDY(n)  ->

          - - (DISPATCH) - ready ((n x.s) e (OUTPUT.c) d.done)
                                  ¬OREQ(n) ¬OBUFRDY(n)

(n x.s) e (OUTPUT.c) d ready done    OREQ(n)  OBUFRDY(n)  ->

          - - (DISPATCH) - ready ((n x.s) e (OUTPUT.c) d.done)
                                   OREQ(n)  OBUFRDY(n)

(n x.s) e (OUTPUT.c) d ready done    OREQ(n) ¬OBUFRDY(n)  ->

   Depending on x:

   x is a recipe [c'.e']

          - - (DISPATCH) - ready (NIL e' c' x
                                  (n x.s) e (OUTPUT.c) d.done)
                                   OREQ(n) ¬OBUFRDY(n)
          and x is altered to be a process cell

   x is a process {}

          - - (DISPATCH) - ready ((n x.s) e (OUTPUT.c) d.done)
                                   OREQ(n) ¬OBUFRDY(n)

   x is a value

          s e c d ready done   OREQ(n)   OBUFRDY(n)   OBUF(n)=x
```

Cycle for polling routine:

Start

↓

Wait until device n ready for output

↓

Set OREQ(n)

↓

Wait until OREQ(n) and OBUFRDY(n) both true

↓

Output contents of OBUF(n), and clear OREQ(n)
and OBUFRDY(n)

The new output driver passes OUTPUT a delayed exploration of the next stream item to be output:

```
λ(out). output(0,out)
whererec output(n,l) ≡ if l=NIL then NIL
                       else (outstream(n,head(l))
                             or output(n+1,tail(l)))
         outstream(n,s) ≡ if s=NIL then NIL
                          else OUTPUT(n,explore(head(s))) ;
                               (NIL or outstream(n,tail(s)))
         explore(x) ≡ if finite(x) then x else UNDEFINED
         finite(x) ≡ . . .
```

where OUTPUT(n,x) compiles:

```
"OUTPUT(n,x)"*m = (LDE x*m|(UPD)) | n*m | (OUTPUT)
```

OXFORD UNIVERSITY COMPUTING LABORATORY
PROGRAMMING RESEARCH GROUP TECHNICAL MONOGRAPHS

SEPTEMBER 1983

This is a series of technical monographs on topics in the field of computation. Copies may be obtained from the Programming Research Group, (Technical Monographs) 8-11 Keble Road, Oxford OX1 3QD, England

PRG-2 Dana Scott
Outline of a Mathematical Theory of Computation

PRG-3 Dana Scott
The Lattice of Flow Diagrams

PRG-5 Dana Scott
Data Types as Lattices

PRG-6 Dana Scott and Christopher Strachey
Toward a Mathematical Semantics for Computer Languages

PRG-7 Dana Scott
Continuous Lattices

PRG-8 Joseph Stoy and Christopher Strachey
OS6 - an Experimental Operating System for a Small Computer

PRG-9 Christopher Strachey and Joseph Stoy
The Text of OSPub

PRG-10 Christopher Strachey
The Varieties of Programming Language

PRG-11 Christopher Strachey and Christopher P Wadsworth
Continuations, A Mathematical Semantics for Handling Full Jumps

PRG-12 Peter Mosses
The Mathematical Semantics of Algol 60

PRG-13 Robert Milne
The Formal Semantics of Computer Languages and their Implementations

PRG-14 Shan S Kuo, Michael H Linck and Sohrab Saadat
A Guide to Communicating Sequential Processes

PRG-15 Joseph Stoy
The Congruence of Two Programming Language Definitions

PRG-16 C A R Hoare, S D Brookes and A W Roscoe
A Theory of Communicating Sequential Processes

PRG-17 Andrew P Black
Report on the Programming Notation 3R

PRG-18 Elizabeth Fielding
The Specification of Abstract Mappings and their Implementation as B^+-trees

PRG-19 Dana Scott
Lectures on a Mathematical Theory of Computation

PRG-20 Zhou Chao Chen and C. A. R. Hoare
Partial Correctness of Communicating Processes and Protocols

PRG-21 Bernard Sufrin
Formal Specification of a Display Editor

PRG-22 C. A. R. Hoare
A Model for Communicating Sequential Processes

PRG-23 C. A. R. Hoare
A Calculus of Total Correctness for Communicating Processes

PRG-24 Bernard Sufrin
Reading Formal Specifications

PRG-25 C. B. Jones
Development Methods for Computer Programs including a Notion of Interference

PRG-26 Zhou Chao Chen
The Consistency of the Calculus of Total Correctness for Communicating Processes

PRG-27 C. A. R. Hoare
Programming is an Engineering Profession

PRG-28 John Hughes
Graph Reduction with Super-Combinators

PRG-29 C. A. R. Hoare
Specifications, Programs and Implementations

PRG-30 Alejandro Teruel
Case Studies in Specification: Four Games

PRG-31 Ian D. Cottam
The Rigorous Development of a System Version Control Database

PRG-32 Peter Henderson, Geraint A. Jones and Simon B. Jones
The Lispkit Manual

PRG-33 C. A. R. Hoare
Notes on Communicating Sequential Processes

PRG-34 Simon B. Jones
Abstract Machine Support for Purely Functional Operating Systems

PRG-35 S. D. Brookes
A Non-deterministic Model for Communicating Sequential Processes

PRG-36 T. Clement
The Formal Specification of a Conference Organizing System