# Interactive functional programs: a method and a formal semantics

Simon Thompson Computing Laboratory, University of Kent at Canterbury, U.K.

July 10, 1992

#### Abstract

In this paper we present a model of interactive programs in a purely functional style. We exploit lazy evaluation in the modelling of streams as lazy lists. We show how programs may be constructed in an ad hoc way, and then present a small set of interactions and combinators which form the basis for a disciplined approach to writing such programs.

One of the difficulties of the ad hoc approach is that the way in which input and output are interleaved by the functions can be unpredictable. In the second half of the paper we use traces, i.e. partial histories of behaviour, to explain the interleaving of input and output, and give a formal explanation of our combinators. We argue that this justifies our claim that the combinators have the intuitively expected behaviour, and finally contrast our approach with another.

# 1 Introduction

This paper explains an approach to describing interactive processes in the lazy functional programming language Miranda<sup>1</sup>[6].

Functional programming is based on *expression evaluation*, where expressions will, in general, contain applications of system- and user-defined functions. Lazy evaluation is a particular strategy for expression evaluation (as there is a *choice* of the way in which we perform the process) which means that

- The values of arguments to functions are only evaluated when they are needed.
- Moreover, if an argument is a composite data object, like a list, the object is only evaluated to the extent that is needed by the function applied to

<sup>&</sup>lt;sup>1</sup> Miranda is a trademark of Research Software Ltd.

it. One of the simplest examples of this is the head function (hd) on lists, which necessitates the evaluation of only the first item in the argument list

The paper consists of two parts. In the first we develop our model of interactive programs in the lazy Miranda system. Since the lazy evaluation strategy is somewhat subtle, we were prompted to try to understand it using formal means — the result is the second half of this paper.

In the first part, after an introductory discussion of lazy evaluation, we introduce the type of interactions and present a number of examples. These are developed in an ad hoc way, which may lead to unexpected interleaving behaviour. We then present a small collection of primitives from which we can build interactions in a disciplined way. We aim to avoid the unexpected by using these primitives alone — despite that we still find there are subtle points which need to be elucidated. This is the purpose of the second part of the paper.

In the second half we aim to give a foundation to the development of interactive processes under the lazy streams approach. We concentrate on a slightly simplified type, which omits the state information included in the first half, since the crucial *interleaving* properties of interactions can be studied more easily in the simpler setting.

After giving a summary of our notation, we introduce, informally at first, the idea of a trace. Traces describe the possible histories of a process (more information about the use of traces to describe processes can be found in [3].) One of the basic ideas in the first part was that of an 'incomplete' or 'partial' interaction — such interactions are the building blocks from which we build full interactive processes. These are re-introduced in section 9 which is followed by a discussion of lazy evaluation and our first attempt at a formal definition of traces. As should be clear, we have tried to motivate the eventual definition of traces by re-tracing our steps towards its final formulation. Section 11 contains a note on the precise nature of print-driven evaluation; in section 12 we introduce the formal definitions of weak pretraces and weak traces, following in section 13 with a number of examples.

To underline the fact that we are dealing with *deterministic* processes, we prove a theorem in section 14 that all the processes described by our model *are* deterministic. The proof is constructive, and shows, incidentally, how we derive the (weak) trace set of a particular process.

Weak traces and pretraces contain sufficient information to describe the behaviour of processes in isolation. Given that we combine processes to build more complex ones, this may not be sufficient, and indeed we show that we have to incorporate more information about termination into our description. This is the aim of section 15, which supplies the definition of pretraces, traces and terminal traces. We revisit our examples in the following section.

A fundamental process combinator is sq, which is intended to perform se- $quential\ composition$  of processes. In section 17 we describe how the trace sets

of two processes are combined sequentially. The subtlety of lazy evaluation which we mentioned above manifests itself here — we see how termination and laziness interact to given an interesting effect when processes are combined: the laziness of reading allows some writing to "overtake" reading. Our definition of sequential composition is justified in section 18 where we prove that sq implements sequential composition as described. This means that we can predict the behaviour of interactive processes when placed in a (sequential) environment, allowing us to write robust and reliable interactive programs in Miranda. Section 19 examines a technical point, raised in the proof of the preceding section.

In the final section we draw some conclusions about the anomalies we found, and the methodology which we recommended, and argue that it does indeed lead to reliable programming. We also contrast our approach with that of the FP/FL school, and finally make some acknowledgements.

We should stress that the trace method adopted here will be applicable to any process (i.e. stream or lazy-list processing program) and not only to interactive processes.

# Part I

# 2 Lazy Evaluation

A feature of a number of modern functional programming languages such as Miranda is that they embody *lazy evaluation*. By this we mean that arguments are passed to functions *unevaluated*. If we look at the function *const* (from the Miranda standard environment) defined thus

$$const \ a \ b = a$$

then the application

$$const (16 + 1) f$$

will return the result without the expression f being evaluated.

An argument is only evaluated if its valued is required by the function.

Suppose we say

$$dconst\ a\ b = a + a$$

and evaluate

$$dconst$$
  $(16+1)$   $b$ 

we get the result 34. In deriving this we may have made no gain, as in evaluating

$$a + a$$

we may have replaced a single evaluation of 16+1 by two such. A naïve approach to demand-driven evaluation might do this — under lazy evaluation we ensure that the result of evaluating an argument is *shared* by all its instances.

A more subtle manifestation of lazy evaluation arises when we consider composite arguments. Once we begin to evaluate a numerical argument, for instance, we evaluate it completely. On the other hand, a composite argument may only be evaluated *partially*. The simplest example is given by the function

$$hd(a:x) = a$$

which returns the first, or head, item a of a list (a:x). If we pass

$$nums$$
 17

to hd when the function nums is defined by

$$nums\ n = n : nums\ (n+1)$$

in order for the application to return the result 17 we only require the fact that nums 17 evaluates partially to

$$17:nums$$
  $18$ 

By the effect illustrated above, lazy evaluation has an effect on the membership of various of the data types of the language, such as *lists*. In the example above we see that the *infinite* list *nums* 17 receives exactly the same treatment as a finite list such as

$$[17, 18, \ldots, 23]$$

In fact, in order for  $hd\ l$  to return 17 all that we need to know about l is that its first member is 17 — the rest, or tail, of the list may be undefined. We usually write  $\bot$  for the undefined list (indeed we write it for the undefined object of any type). Using this notation we see that

$$hd(17:\bot) = 17$$

so *partial* lists, which have undefined final segments, are legitimate lists in our lazy scheme of things.

Input and output are often thought of as *streams*. Given the discussion above we can see that streams can be identified with lazy lists.

• The operation of testing whether a stream contains an item corresponds to pattern matching the list with the pattern (a:x). In case this is successful, the item will be bound to a and the remainder of the stream to x.

 $\bullet$  On the other hand, given an item b, the stream

is a stream whose first element is b and whose remainder consists of y. We can thus view the list construction operation ":" as an output operation, placing an item onto a stream.

What is the effect of evaluating an expression such as

nums 17

or

nums' 17 100

when

$$nums' \ n \ m = []$$
 ,  $n > m$   
=  $n : nums' (n + 1) m$  ,  $otherwise$ 

Output will be produced in an incremental fashion: first the first element of the list will be evaluated and printed, then the second and so on. Portions of the output will be printed before the evaluation is complete. In particular, if the result being printed is a function application, the need to print will drive the evaluation of the arguments to the function. As we hinted above, lazy evaluation can be seen as a *species* of demand-driven dataflow. We discuss this latter fact in the context of interactive programming in section 10 below.

# 3 A type of interactions

An interactive program is designed to read from a stream of input, and to write to a stream of output. As we have already observed, we can view streams as *lists*, so if we say

$$input == [char]$$
 $output == [char]$ 

then the type of functions

$$input \rightarrow output$$

forms a simple model of interactive processes. For instance, a process which will double space its input can be written thus:

```
double \perp space \ (a:x) = a: double \perp space \ x , a \sim = newline 
= a: a: double \perp space \ x , otherwise 
double \perp space \ [] = []
```

and a process which simply copies (or echoes) its input is written

$$echo y = y$$

The equations we have just supplied describe how the output stream depends upon the input stream. In an interactive context we are likely to be interested not only in the input/output relation but also in the way that the two streams are *interleaved* temporally, for example on a terminal screen. Recall that our lists are lazy, and our discussion of the way in which lists are printed. We mentioned in that account that output will begin to be produced as soon as is possible, and that further output will be generated similarly, contingent upon the presence of sufficient input. In printing the result of

#### $echo\ stdin$

(stdin denotes standard input), a character can be echoed as soon as it is typed at the terminal, since an item will be placed on the output stream as soon as it appears on the input stream. (Users of 'real' systems will perhaps observe something different, as most terminals will buffer their input into lines before transmitting it to a host. In such a situation echoing will happen as promptly as possible, i. e. line by line.)

In most cases laziness has the effect which we would intend. Nonetheless, it can have some unpredicted effects. We return to this in section 6.

Before we continue we should emphasise that our model is sufficiently powerful to capture processes which have an *internal state*. A particular item on the output stream will depend on the whole of the input stream which has thus far been read, and so will depend on the whole *history* of the input to the process. Just to give a brief example, we can write a program which either single or double spaces its input, where \$ is used to toggle between the two modes. The function, which we call  $option \perp on$ , starts off in double spacing mode.

```
\begin{array}{rcll} option \bot on \; (a:x) & = & option \bot off \; x & , a = '\; \$' \\ & = & a:a:option \bot on \; x & , a = newline \\ & = & a:option \bot on \; x & , otherwise \\ option \bot off \; (a:x) & = & option \bot on \; x & , a = '\; \$' \\ & = & a:option \bot off \; x & , otherwise \end{array}
```

# 4 Partial Interactions

An interactive process in isolation is specified by a function of type

$$input \rightarrow output$$

which describes the form of the output in terms of the input. However in general we wish to combine simple interactions into composite ones. If we think of following one interaction by another, we need to be able to pass the portion of the input stream unexamined by the first on to the second. Such partial interactions must therefore return the unconsumed portion of the input stream

as a part of their results. These partial interactions will therefore be of type

$$input \rightarrow (input, output)$$

Consider the example of an interaction which reads a line of input and outputs its length.

```
line \perp len :: input \rightarrow (input, output)
line \perp len in
= (rest, out)
where
out = show (\# line)
(line, rest) = get \perp line [] in
get \perp line \ (front \ (a : x))
= get \perp line \ (front \ ++ \ [a]) \ x \quad , a \sim = new line
= (front, x) \quad , otherwise
```

 $get \perp line$  is used to get the first line from the input stream. It returns a result consisting of the line paired with the remainder of the input, rest. The first parameter of  $get \perp line$  is used to accumulate the partial line, and show is a function converting a number to a printable form.

We need to make one further refinement to the model. As we are now contemplating building interactions from simpler components, we may want to pass (state) information from one interaction to another. In general we think of an interaction as being supplied with a value, of type \* say, on its initiation and returning a value of a possibly different type, \*\* say, on termination. This gives a general type of partial interactions

$$interact * ** == (input, *) \rightarrow (input, **, output)$$

To summarise

- Interactions are modelled by a function type, parametrised on two type variables \*, \*\*.
- The domain type is (input, \*) items of this type are pairs consisting of
  - input streams and
  - initial state values.
- The range type is (input, \*\*, output) items from which are triples, consisting of
  - the portion of the input stream unexamined by the interaction,
  - the final state value, and
  - the output produced during the interaction.

There are natural examples of interactions for which \* and \*\* are different. For instance, if  $get \perp number$  is meant to get a number from the input stream then its natural type would be

```
interact () num
```

() is the one element type, whose single member is (), the empty tuple — its use here signifies that no prior state information is required by the process.

To give an example of an interaction this type, we might consider modifying  $line \perp len$  so that it will print an accumulated total number of characters after each line, as well as the length of the line itself.

```
line \perp len \perp deluxe :: interact num num
```

```
\begin{array}{lll} line \bot len \bot deluxe \ (in,tot) \\ &= \ (rest,newtot,out) \\ &where \\ &(line,rest) &= get \bot line \ [\,] \ in \\ &len &= \# \ line \\ &newtot &= tot + len \\ &out &= show \ len + + show \ newtot \end{array}
```

The state information passed in by the interaction is modified by the addition of the current line length.

# 5 Combining Interactions

Up to this point we have considered 'primitive' interactions, built in an ad hoc way. In this section we look at some functions which enable us to combine interactions in a disciplined way, with the consequence that their interactive behaviour will be more predictable.

First we introduce a number of basic interactions and then we present some combining forms or *combinators* which build complex interactions from simpler ones.

## 5.1 Basic Interactions

First, to read single characters we have

```
get \perp char :: interact * char
get \perp char ((a:x), st) = (x, a, [])
```

and to write single characters,

```
put \perp char :: char \rightarrow interact **

put \perp char \ ch \ (in, st) = (in, st, [ch])
```

We can also perform 'internal' actions, applying a function to the state value:

$$apply :: (* \rightarrow **) \rightarrow interact * **$$
 $apply f (in, st) = (in, f st, [])$ 

These are three atomic operations from which we can build all our interactions using the combinators which follow. That these are sufficient should be clear from the fact that they give the atomic operations on input, output and internal state, respectively.

# 5.2 Sequential Composition

The type of the sequential composition operator sq is

```
interact * ** \rightarrow interact ** *** \rightarrow interact * ***
```

sq first second should have the effect of performing first and then second, so

```
\begin{array}{lll} sq\ first\ second\ (in,st) \\ &= & (rest,final,out\bot first\ ++\ out\bot second) \\ &\quad where \\ &\quad (rem\bot first,inter,out\bot first) &= first\ (in,st) \\ &\quad (rest,final,out\bot second) &= second\ (rem\bot first,inter) \end{array}
```

first is applied to (in, st) resulting in output  $out \perp first$ , new state inter and with  $rem \perp first$  the remainder of the input. The latter, paired with inter, is passed to second, with result

```
(rest, final, out \perp second)
```

The input remaining after the composite action is rest, the final state value is final and the overall output produced is the concatenation of the output produced by the individual processes,

```
out \perp first ++ out \perp second
```

and so we return the triple of these values as the result of the combination. We explore the precise interleaving behaviour of this combinator in the second part of this paper.

## 5.3 Alternation and Repetition

To choose between two alternative interactions, according to a condition on the initial state, we use the alt combinator, which is of type

```
cond * \rightarrow interact * ** \rightarrow interact * ** \rightarrow interact * **
```

```
alt\ condit\ inter1\ inter2\ (in,st) = inter1\ (in,st) , condit\ st
= inter2\ (in,st) , otherwise
```

 $cond * == * \rightarrow bool$  is the type of predicates or conditions over the type \*. The effect of alt is to evaluate the condition on the input state, condit st, and according to the truth or falsity of the result choose to invoke the first or second interaction.

The trivial interaction

$$skip :: interact \ * \ *$$

does nothing

$$skip(in, st) = (in, st, [])$$

(Observe that we could have used apply to define skip since it is given by apply id). Using sq, alt, skip and recursion we can give a high level definition of iteration:

$$while :: cond * \rightarrow interact * * \rightarrow interact * *$$

which we define by

```
while condit inter
= loop
where
loop = alt condit (inter $sq loop) skip
```

'Depending on the condition, we either perform inter and re-enter the loop or we skip, i.e. do nothing to the state and terminate forthwith.' Note, incidentally, that we have prefixed sq by \$ to make it an infix operator. Using while we can define a repeat loop:

```
repeat :: cond * \rightarrow interact * * \rightarrow interact * *

repeat condit inter

= inter \$sq \ (while \ not \bot condit \ inter)

where

not \bot condit = (\sim).condit
```

 $\sim$  is the boolean negation function, so that  $not \perp condit$  is the converse of the condit condition.

# 5.4 Using the combinators

In this section we give an example of a full interaction, that is an interaction of type

$$input \rightarrow output$$

which is built from partial interactions using the combinators. The program inputs lines of text repeatedly, until a total of at least one thousand characters has been input, at which point it halts. After each line of input the length of the line and the total number of characters seen thus far is printed. Define the numerical condition

$$sufficient \ n = (n >= 1000)$$

now if

```
rep \perp inter :: interact\ num\ num

rep \perp inter = repeat\ sufficient\ line \perp len \perp deluxe
```

we can define our full interaction,

$$full \perp inter :: input \rightarrow output$$

by

$$full \perp inter in$$

$$= out$$

$$where$$

$$(rest, final, out) = rep \perp inter (in, 0)$$

It should be obvious why we have chosen the starting value for the state to be zero, since no characters have been read on initiation of the process.

# 6 Two Cautionary Examples

We mentioned that interaction functions that we define may not always behave as we expect. The two examples we present here illustrate two different ways in which that can happen.

## 6.1 Pattern Matching

Pattern matching can delay output. We might write a function which prompts for an item of input and then echoes it thus:

$$try(a:x) = "Prompt:" ++ [a]$$

Unfortunately, the prompt will only be printed after the item has been input. This is because the evaluator can only begin to produce output once that match with the pattern (a:x) has succeeded, and that means precisely that the item has entered the input stream. We can achieve the desired effect by writing

$$again x = "Prompt : " ++ [hd x]$$

The prompt will appear before any input has been entered, as nothing needs to be known about the argument (x) for that portion of the output to be printed.

# 6.2 Lazy Reading

Consider the process

while (const True) get

where

get :: interact \* \*

is defined by

$$get(in, st) = (tl\ in, st, "Prompt:")$$

What is the effect? We first envisage that it repeats the interaction get indefinitely, and the effect of get is to prompt for an item input and then to read it. In fact we see that the prompt is printed indefinitely, and at no stage does input take place. As we explained above, output is driven by the need to print, and the output from this interaction can be derived without any information about the input stream, with the consequent effect that no input is read.

The second example, of lazy reading, causes a major headache. We shall see presently precisely how writes can overtake reads, and in the conclusion to the paper we argue that this will not happen under the disciplined approach advocated here.

# 7 Miscellany and Conclusions

We have shown how interactive programs can be written in a disciplined way in a functional system. Central to this enterprise are

- streams, implemented here as lazy lists, but available in other languages as objects distinct from lists, and
- higher order functions. The type *interact* \* \*\* of interactions is a function type, and so our interaction combinators are inescapably higher order.

We need not be limited to sequential combinators in our definitions. We can, for example, think of resetting a state to its initial value after an interaction, which means that, for instance, we can perform a "commutative" or "pseudo-parallel" composition of processes. Such combinations of processes can be useful when we write input routines for structured objects.

We can view the type *interact* \* \*\* in a slightly different way. We can see the type as one of *functions* which read input and produce output: these pseudo-functions are from type \* to \*\*, and they return as part of their results the input stream after their application, together with the output produced. This gives us another perspective on the combinators defined above.

Observe also that not every member of the type interact \* \*\* is a natural representative of an interaction. All the interactions <math>f we have seen have the property that if

$$f(in, st) = (rest, st', out)$$

then rest will be a final segment of in (i.e. will result from removing an initial portion from in).

This seems to be the place to make a polemical point. Much has recently been made of the notion of 'multi-paradigm programming'; this work can be seen as an antidote to this. We have seen that the functional paradigm will allow us to model another paradigm (the imperative) in a straightforward way, without sacrificing the elegant formal properties of the functional domain. A naïve combination of the two sacrifices the power and elegance which each possesses individually.

In the next part of the paper we explore a formal *trace* semantics for interactions, in order to resolve any difficulties we may have had with interleaving.

# Part II

# 8 Notation

Here we introduce notation we shall use in the remainder of the document. The reader may wish to skip the section on first reading and refer back to it if and when it is necessary.  $tr, tr', \ldots$  will range over sequences, that is finite lists which are terminated by [] and not by  $\bot$ . We say

$$tr \subset tr'$$

if tr is an initial segment of tr', i.e.

$$\#tr \leq \#tr'$$

(# gives the length of a list) and

$$\forall n < \#tr . tr! n = tr'! n$$

We write

$$tr \subset tr'$$

if and only if  $tr \subset tr'$  and  $tr \neq tr'$ .

Most of our sequences will consist of objects of three kinds. These will be input objects, tagged with r (for read), output objects, tagged with w (for write) and the object  $\sqrt{}$ , which we use to indicate termination — we shall explain this further in the body of the paper. We write

$$tr \subset_c tr'$$

if and only if  $tr \subset tr'$  and the element which follows the initial segment tr in tr' is not an output object (tagged with w) — we call these initial segments complete because they are complete with respect to writing.

Often we wish to look at the input or output portions of the sequences separately. We write

$$tr \ \lceil \ in$$

$$tr \lceil out$$

for these restrictions. For example, if

$$tr = [r2, r3, w5, \sqrt{r7}]$$

then

$$tr [in = [2, 3, 7]]$$

$$tr [out = [5]$$

If f is a function of type

$$input \rightarrow (input, output)$$

then we write

$$f insq \mapsto outsq$$

if

$$f (insq ++ \perp)$$

is equal to one of

$$\begin{array}{c} (\bot, outsq ++ \bot) \\ (\bot, outsq) \\ \bot \end{array}$$

with the latter possibility only in the case that outsq = []. We write

$$f insq \downarrow outsq$$

in the case that

$$f(insq ++ \bot) = (\bot, outsq)$$

We use a number of standard list functions in the sequel. Each non-bottom list is of the form [] or (a:x) for some head element a and tail list x. We write our definitions using pattern matching over these cases:

$$\begin{array}{rcl} hd\left(a:x\right) & = & a \\ tl\left(a:x\right) & = & x \\ hd\left[\right] & = & \bot \\ tl\left[\right] & = & \bot \end{array}$$

Finally, note that we use ++ as list (or sequence) concatenation, so

$$[3, 4] ++ [5, 6] = [3, 4, 5, 6]$$

## 9 Traces

Our aim in this document is to describe the semantics of interactive programs by means of *traces* of their behaviour. These traces will be sequences of actions which can be of three kinds

r a is a read, or input, action — the item a is read from the input stream

w a is a write, or output, action — the item a is written to the output stream

 $\sqrt{}$  is an (invisible) action which signals that the process has completed its output — we shall say some more about this below, including an discussion of why we need such an action in our descriptions.

We say that a sequence tr is a trace of a process P if and only if the sequence of actions in tr is a possible behaviour of the process P. For example, a process which

- prompts for an item by writing a prompt,
- reads an item from the input stream and
- echoes the item read

will have

$$[w \ prompt, r \ a, w \ a]$$

as one of its traces. In fact, if the process terminates after echoing the item read, the full set of traces will be

 $[ ] \ [w\ prompt] \ [w\ prompt, r\ a] \ [w\ prompt, r\ a, w\ a] \ [w\ prompt, r\ a, w\ a, \sqrt{]}$ 

Further details of a calculus of processes and their associated trace semantics can be found in [3]. We shall use the set of traces of a process as the means by which we *specify* a process and our aim in this paper will be to describe Miranda processes by means of their trace sets. There is an associated problem of describing these sets of traces, one solution to which may be provided by temporal logic, [4]. This is not something which we address here.

How do we model interactive programs in Miranda? As we saw above, we consider the programs to be mappings between the *streams* of input and output. We model a complete interaction as a function

$$input \rightarrow output$$

where input and output are lists of items. The interaction we specified above is described by the function

$$example_1 \ x = [prompt, hd \ x]$$

Note that

$$[w \ prompt, r \ a, w \ a, \sqrt] \ [out = [prompt, a]$$

and

$$[w \ prompt, r \ a, w \ a, \sqrt{}] \ [in = [a]]$$

and that the portion of the input read, from a list (a:x) will be [a], so that the trace

$$[w \ prompt, r \ a, w \ a, \sqrt{}]$$

is an *interleaving* of the input consumed and output written by the function  $example_1$  (together with the termination information given by  $\sqrt{\ }$ ). We discuss the formal means by which we find the traces of functional interactions below.

In the first part we saw that in general we need to consider *partial* interactions, *i.e.* objects of type

$$interact == input \rightarrow (input, output)$$

which return unconsumed input as a part of their result. Our model of the example interaction above is now

$$example_2 x = (tl x, [prompt, hd x])$$

where we see that the tail of the list x, tl x is the remainder of the input stream, passed to a succeeding interaction, if any.

In the first part we also added state information to the model; we do not do that here, as our purpose is to concentrate on input/output behaviour.

Recall that we combine two interactions sequentially thus:

$$sq::interact \rightarrow interact \rightarrow interact$$

$$sq\ inter_1\ inter_2\ in = (rest, out_1 ++ out_2)$$
 
$$where$$
 
$$(betw, out_1) = inter_1\ in$$
 
$$(rest, out_2) = inter_2\ betw$$

How do we explain this? The output of the combined process is the second component of the pair, and is the concatenation of the outputs  $out_1, out_2$  produced respectively by the processes  $inter_1$  when supplied with input in and  $inter_2$  when supplied with input betw (for between), the input remaining after the first interaction. We shall give a formal justification for this explanation later.

# 10 Lazy Evaluation Revisited

An expression written in the Miranda language has its evaluation driven by the need to write or produce the result — evaluation is demand driven, and function arguments are only evaluated if and when they need to be. Such a scheme is called lazy evaluation. When arguments and results are structured, in particular when they are lists, laziness means that

- The component parts of lists are written as soon as they are available, so we can see writing is eager, in a sense.
- The component parts of an argument list are only read as they are needed.

Consider the example of  $example_1$  from section 9. The first item of the output list, prompt, can be written without any examination of the input list x. On the other hand, the second item is the head of the input list, and so this item needs to be read before writing can proceed further. Once it is read it can be output, with no further examination of the input list.

This analysis justifies our claim that

$$[w \ prompt, r \ a, w \ a]$$

is a trace of the function  $example_1$ . How could we show this in a rigorous way? We do this by means of the denotational semantics of our language, [2, 5]. In particular we analyse the behaviour of our function on partial lists — lists which are terminated by the bottom element  $\bot$ .  $\bot$  represents the state of our knowing nothing about a value, so that the partial list

$$[2,3]++\bot$$

represents a list about which we know nothing except its first two elements.

What is the semantics of  $example_1$ ? Recall the definition of hd (from section 8) and note that

$$hd \perp = \perp$$

(This should be an obvious truth — we can deduce it from the monotonicity of the semantic interpretation, which we shall discuss further below.) Now,

$$\begin{array}{rcl} example_1 \perp & = & [prompt, hd \perp] \\ & = & [prompt] + + \perp \\ \\ example_1([a] + + \perp) & = & [prompt, hd ([a] + + \perp)] \\ & = & [prompt, a] \end{array}$$

We see, in the two cases, that the input sequences [], [a] give rise to the output sequences [prompt], [prompt, a]. (We shall have something more to say about

these functions in section 11). If we write  $\sim$  instead of 'gives rise to', we can make our first attempt at defining traces:

**Definition attempt 1:** We say that tr is a trace of f if for each initial segment tr' of tr,

$$f tr' [in \leadsto tr' [out]$$

This definition contains the essence of the definition we shall end up with in section 12. Our definition states that

$$f tr [in \leadsto tr [out]]$$

and that moreover this holds for every initial segment of tr. Clearly we need this second condition, since any merge  $tr_1$  of the sequences  $tr \lceil in, tr \rceil out$  will satisfy

$$tr_1 \lceil out = tr \lceil out$$
  
 $tr_1 \lceil in = tr \lceil in$ 

It is not hard to see that asking for the property to hold of every initial segment tr' of tr means that we are asking for a particular interleaving of the sequences — that embodied by tr. This is reflected in our definition and theorem on determinacy in section 14. In section 12 we refine our attempted definition, but first we say something about approximation and monotonicity. The ordering on the domain of interpretation,  $\sqsubseteq$ , is an ordering of increasing information. For example, for every x,

$$\perp \sqsubseteq x$$

and for two finite sequences s, t,

$$s \subseteq t$$

if and only if

$$s + + \perp \sqsubseteq t + + \perp$$

The interpretation of this equivalence is revealing. Extending the defined portion of a stream, from s to t, gives more information about the stream. Conversely, more information about partial streams terminated by  $\bot$  is given by extending the defined portion, that is by providing more input items.

We say that a function is monotone if for all x, y,

$$x \sqsubset y \Rightarrow f x \sqsubset f y$$

in words, if we give more information about the argument, we can only get more information about the result.

We should note that for finite sequences s,t  $s \sqsubseteq t$  if and only if s=t. This is because the lists are *definite* — they cannot be further extended as they are fully defined. Note however that

$$s + + \perp \sqsubseteq s$$

for every s.

The reader should have no difficulty now in seeing that the only possible value for  $hd \perp$  (and indeed for  $tl \perp$ ) must be  $\perp$ .

# 11 A note on printing

In our discussion in the previous section, we stated that

$$example_1 \perp = [prompt, hd \perp]$$
  
=  $[prompt] + + \perp$ 

the second equality is *not* in fact true! We shall see here that as far as the printing device, or evaluator, is concerned they are equivalent.

How does the printer print values, when those values are lists? The list is printed one item at a time, starting from the head. If at any point one of the items of the list is not (so far) defined, then no further output will be produced. We can write this as a Miranda-like function thus:

$$print'[] = []$$
  
 $print'(a:x) = a:print'x, a \neq \bot$   
 $= \bot, otherwise$ 

As this definition stands, it is not computable, but if we replace the test  $a \neq \bot$  with the test for (computable) self-equality a = a, we get the same effect:

$$print[] = []$$
  
 $print(a:x) = a:print x, a = a$ 

Let us now examine the effect of printing  $[prompt, \bot]$  according to the print function:

$$print [prompt, \bot] = prompt : print [\bot]$$
  
=  $prompt : \bot$ 

since the evaluation of the guard  $\perp = \perp$  is non-terminating. This final expression is certainly equal to

$$[prompt] ++ \bot$$

and so we justify our sleight of hand in the previous section. For the rest of the paper we shall assume that our functional interactions produce output which is "printable" directly; we can ensure this by composing them with *print* if we wish. Note, of course, that printable output will take the form

$$outsq\\ outsq ++ \bot\\ infsq$$

where outsq is a (finite) sequence, and infsq is an infinite sequence.

Note that we can have two denotationally unequal functions whose printable behaviour will be equivalent. Consider the example function

$$example_1 \ x = [prompt, hd \ x]$$

and the variant

$$\begin{array}{rcl} example_3 \; x & = & [prompt] + + \; out x \\ & & where \\ & out \; (a:z) = a \end{array}$$
 
$$\begin{array}{rcl} example_1 \; \bot & = & [prompt, \bot] \\ example_3 \; \bot & = & [prompt] + + \; \bot \end{array}$$
 
$$\begin{array}{rcl} example_1 \; [] \; = & [prompt, \bot] \\ example_3 \; [] \; = & [prompt] + + \; \bot \end{array}$$
 
$$\begin{array}{rcl} example_1 \; (a:z) \; = \; [prompt, a] \\ example_3 \; (a:z) \; = \; [prompt, a] \end{array}$$

As we saw above, print gives the same result on the outputs of the example where they are different, so that the printable versions of the functions are exactly the same.

# 12 Weak pretraces and traces

We made our first attempt at a definition of the trace of a complete interaction on page 18. In this section we refine the definition in the light of our example,  $example_3$ , which we saw in the previous section, and also adapt it to partial interactions, i.e. objects of type interact.

We saw in the earlier section that the example has the same printing behaviour as  $example_1$ , which we already know has traces

```
[w \ prompt][w \ prompt, r \ a][w \ prompt, r \ a, w \ a]
```

Do each of these sequences have the property that

$$example_3 sq [in \rightsquigarrow sq [out ?]]$$

We work through them in turn.

$$example_3([]++\bot) = [prompt]++\bot$$
  
 $example_3([a]++\bot) = [prompt,a]$ 

Interpreting these, we have that

$$example_3 [] \sim [prompt]$$
  
 $example_3 [a] \sim [prompt, a]$ 

so that we have

$$example_3 tr' [in \leadsto tr' [out]]$$

only for the sequences  $[w \ prompt]$  and  $[w \ prompt, r \ a, w \ a]$ , and not for the other two initial segments of  $[w \ prompt, r \ a, w \ a]$ , that is [] and  $[w \ prompt, r \ a]$ . Why should this be? Remember that lazy evaluation is used to implement the language, and as we observed in section 10, this means that  $writing \ is \ eager$ . The input portion of the trace

$$[w \ prompt, r \ a]$$

is sufficient to provoke the writing of the item a. This fact is not registered by the initial segment  $[w \ prompt, r \ a]$ , as it is not "complete" with respect to writing. We say that a proper initial segment tr' of tr is complete,

$$tr' \subset_c tr$$

if and only if  $tr' \subset tr$  and the element following tr' in tr is not a w element. As we see in the example above, we can only expect the property

$$f tr' \lceil in \leadsto tr' \lceil out \rceil$$

for complete initial segments of a potential trace.

If we consider the process described by example<sub>3</sub> as a partial interaction, we should record the input consumed by the interaction. We saw in section 9 that we could do this by returning the remainder of the input as one component of the result, the other being the (partial) output. The natural way that we modify the example function is

$$example_3 x = (tl x, [prompt] ++ out x)$$

$$where$$

$$out (a:z) = a$$

This registers the fact that the interaction consumes only the first item of the input stream. Now, we define the relation "gives rise to" thus:

## **Definition:**

$$f insq \mapsto outsq$$

if and only if

$$f (insq ++ \perp)$$

is equal to

$$(\perp, outsq ++ \perp)$$

or

$$(\perp, outsq)$$

or in the case that outsq is  $\perp$ ,

⅃

In each of these cases, we register the fact that the portion of input presented, insq, is read fully by requiring that the remainder of the input returned is  $\bot$ —if an item in insq had not been read then it would form part of the sequence returned.

# **Definition** of weak trace and pretrace

tr is a weak pretrace of f if and only if f  $tr \lceil in \mapsto tr \lceil out$  and for every complete initial segment tr' of tr

$$f tr' [in \mapsto tr' [out]$$

tr is a weak trace of f if and only if it is an initial segment of a weak pretrace of f.

(Aside If we look at our previous definition of  $example_3$  and of traces we shall see that

$$f insq \sim [a]$$

for any insq which begins with a — no account is taken by this definition that the input is not read beyond the first item.)

We shall examine the definition of weak traces and give the full definition of traces in section 9. First we examine some examples in detail.

# 13 The weak traces of some examples

In this section we look at a number of example interactions, describing their weak traces and pretraces.

## Example 1

$$write in = (in, [message])$$

We want to find the sequences insq and outsq such that

$$write insq \mapsto outsq$$

Note first that

$$write x = \bot$$

for no x. Now,

write 
$$x = (\bot, y)$$

if and only if  $x = \bot$  and y = [message], so

$$write [] \mapsto [message]$$

This means that  $[w\ message]$  is a candidate weak pretrace, if all its complete initial segments have the same property.  $[\ ]$  is the only proper initial segment, and this is not complete, so the condition is satisfied vacuously. The set of weak traces will be

$$\{[], [w \ message]\}$$

# Example 2

$$read(a:x) = (x, [])$$

Now, observe that

$$read y = \bot$$

if y is [] or  $\bot$ . Because of the latter possibility,

$$read[] \mapsto []$$

Now,

$$read(a:x) = (\bot, y)$$

if and only if  $x = \bot$  and y = [], so,

$$read[a] \mapsto []$$

Now, we can see that [] is a weak pretrace, and  $[r\ a]$  is a possible pretrace. Its only complete initial segment is [], and this has the required property, so we have two weak pretraces,

$$\{[\ ], [r\ a]\}$$

which are exactly the weak traces too.

# Example 3

$$echo(a:x) = (x,[a])$$

$$echo\ y = \bot$$

if 
$$y = []$$
 or  $\bot$ , so

$$echo[] \mapsto []$$

making [] a weak pretrace. Now,

$$echo\ y = (\bot, z)$$

if and only if  $y = a : \bot$  and z = [a], so

$$echo[a] \mapsto [a]$$

which means that potentially

$$tr_1 = [w \ a, r \ a]$$

$$tr_2 = [r \ a, w \ a]$$

are weak pretraces.  $tr_2$  has only [] as complete initial segment, and by our remark above that [] is a weak pretrace,  $tr_2$  will be a weak pretrace.  $tr' = [w \ a]$  is a complete initial segment of  $tr_1$ , but it is not the case that

$$echo\ tr'\ [\ in\ \mapsto tr'\ [\ out$$

so  $tr_1$  will not be a weak pretrace. The set of traces will be

$$\{[\ ], [r\ a], [r\ a, w\ a]\}$$

We begin to see the subtlety of the analysis in the contrast between the following examples.

## Example 4

$$promptin_1 (a:x) = (x, [prompt])$$

$$promptin_1 \ y = \bot$$

if  $y = \bot$  or [], implying that

$$promptin_1[] \mapsto []$$

making [] a weak pretrace. Similarly to the previous example,

$$promptin_1 [a] \mapsto [prompt]$$

and as above, the possible pretraces are

$$tr_1 = [r \ a, w \ prompt]$$
  
 $tr_2 = [w \ prompt, r \ a]$ 

 $tr_1$  is a weak pretrace, since its only complete initial segment is [].  $tr_2$  fails to be — this is a surprise, as we might expect the prompt to precede the input of the item. The delay is caused by the fact that the right hand side is only produced after a successful pattern match, and this can only take place after an input item is present. We should contrast this with the next example.

#### Example 5

$$promptin_2 \ y = (tl \ y, [prompt])$$

First note that

$$promptin_2 \ y \neq \bot$$

for all y.

$$promptin_2 \ y = (\bot, z)$$

if and only if  $tl\ y = \bot$  and z = [prompt]. In turn,  $tl\ y = \bot$  if y is  $\bot$ ,  $a:\bot$  or []. The first two eventualities mean that

$$promptin_2[] \mapsto [prompt]$$
  
 $promptin_2[a] \mapsto [prompt]$ 

The first of these means that

$$[w \ prompt]$$

is a weak pretrace, but which of

$$tr_1 = [r \ a, w \ prompt]$$
  
 $tr_2 = [w \ prompt, r \ a]$ 

will be? Now,  $tr_1$  has [] as a complete initial segment, but [] is not a weak pretrace, so  $tr_1$  is not either. On the other hand, the complete initial segment  $[w \ prompt]$  of  $tr_2$  is a weak pretrace, so  $tr_2$  will be a weak pretrace too, giving the trace set

$$\{[\ ], [w\ prompt], [w\ prompt, r\ a]\}$$

We can see from the examples that we have presented that we can divine the behaviour of the interactions by analysing their implementing functions over partial lists.

We shall see in section 15 that the set of weak traces is insufficient to describe the behaviour of one of these interactions when embedded in a context. First we look at a result on the determinacy of our functional interactions.

# 14 Determinacy

Using the functional approach, we write interactive processes as functions

$$input \rightarrow (input, output)$$

Functions associate their results with their arguments in a deterministic way – this carries over to our functional processes, a result which we formulate and prove in this section.

Theorem 1 (Determinacy) For all sequences insq, outsq if

$$f insq \mapsto outsq$$

there is a unique weak pretrace tr of f with  $tr \lceil in = insq$  and  $tr \lceil out = outsq$ .

In other words, if an input sequence insq gives rise to an output sequence outsq, then there is a unique interleaving tr of the two which forms a behaviour of the process — the order in which the actions of input and output take place is determinate.

#### Proof:

We first construct our weak pretrace and then prove its uniqueness. Before we do that, we look at how initial segments of insq are treated by f. Suppose that  $sq \subseteq insq$ , then  $sq ++ \bot \sqsubseteq insq ++ \bot$ , so by monotonicity one of the following will hold:

$$\begin{array}{ccc} f \; (sq \; ++ \; \bot) & \sqsubseteq & (\perp, outsq) \\ f \; (sq \; ++ \; \bot) & \sqsubseteq & (\perp, outsq \; ++ \; \bot) \\ f \; (sq \; ++ \; \bot) & \sqsubseteq & \bot \end{array}$$

depending on the reason that  $f insq \mapsto outsq$ . In any case

$$f(sq ++ \bot) \sqsubseteq (\bot, outsq)$$

Recall the assumption we made in section 11 on the "printed form" of the output. Using this assumption, we can conclude that  $f(sq ++ \bot)$  is equal to one (and only one) of

$$(\bot, sq') \tag{1}$$

$$(\perp, sq' + + \perp) \tag{2}$$

$$\perp$$
 (3)

for some  $sq'\subseteq outsq$ . We reiterate: for all  $sq\subseteq insq$  there exists a unique  $sq'\subseteq outsq$  such that

$$f sq \mapsto sq'$$

We use the notation  $f \triangleright sq$  for this sequence sq'.

We now build the weak pretrace, by examining the output sequences given by the initial segments of insq. Take  $sq = [\,\,] \subseteq insq$  first. The partial trace p is set to  $f \triangleright sq$ . Now, in the general case, consider the initial segment extended by one element, a say. Call the new segment sq. If (3) holds we simply append  $(r \ a)$  to the partial trace. If (1) or (2) holds, we first append  $(r \ a)$  to the partial trace, and then append to the result the elements "newly written". This is the sequence of elements by which  $f \triangleright sq$  extends  $p \lceil out$ , in other words the final segment of  $f \triangleright sq$  resulting from the removal of the initial segment  $p \lceil out$ .

At each stage we produce a complete initial segment of the final sequence, and moreover each such will have the weak pretrace property. We have therefore produced the weak pretrace which we call the *canonical* one.

Now, suppose that there exists another interleaving of the sequences, tr, which is also a weak pretrace. Choose the maximal common initial segment of tr and canon and call it tr'. Since the elements following tr' in tr and canon are different, and the two are interleavings of the same sequences, tr' must be followed by a read in one, and a write in the other. This makes it a complete pretrace of the former, and so

$$f tr' [in \mapsto tr' [out$$

On the other hand, in the latter sequence, tr' is followed by a write, so by the weak pretrace property there is an outsq',  $tr' \mid out \subset outsq'$  with

$$f tr' [in \mapsto outsq']$$

This contradicts the single-valuedness, or determinacy, of the function f, so allows us to conclude that canon is the unique interleaving forming a weak pretrace.  $\Box$ 

## 15 Traces

We have seen how the behaviour of an interaction in isolation can be described by the collection of weak traces. In this section we look at an example which exposes the limits of this approach, and then introduce the full definition of traces, which extend the descriptive information available.

We start with our example.

$$read_1 (a:x) = (x,[])$$
  
 $read_2 y = (tl y,[])$ 

Both these processes read a single item from the input stream, and produce no output. Examining the formal details we see that

$$read_1 \perp = \perp$$

$$read_1 (a : \bot) = (\bot, [])$$
  
 $read_2 \bot = (\bot, [])$   
 $read_2 (a : \bot) = (\bot, [])$ 

which means that each has the set of traces

$$\{[\ ], [r\ a]\}$$

How do the two interactions differ? If we interpret

$$read_2 \perp = (\perp, [])$$

we can see that the process produces no output, and the moreover termination of output happens before any input is read. Indeed if we follow the interaction by a process which writes,

$$write y = (y, [message])$$

then the message will be output before any input is read. On the other hand, if we follow  $read_1$  with write, reading precedes writing. We justify these assertions by examining the compositions

$$rw_1 = sq \ read_1 \ write$$
  
 $rw_2 = sq \ read_2 \ write$ 

where we use sq as defined in section 9 to compose the processes.

$$rw_2 \perp = (rest, out_1 ++ out_2)$$
 $where$ 
 $(betw, out_1) = read_2 \perp$ 
 $(rest, out_2) = write\ betw$ 

Now,

$$\begin{array}{rcl} (betw, out_1) & = & (\bot, [\,]) \\ (rest, out_2) & = & (\bot, [message]) \end{array}$$

 $\mathbf{so}$ 

$$rw_2 \perp = (\perp, [message])$$

which implies that

$$rw_2 [] \mapsto [message]$$

which means that writing precedes reading in this case.

On the other hand,

$$rw_1 \perp = (rest, out_1 ++ out_2)$$
 $where$ 
 $(betw, out_1) = read_1 \perp$ 
 $(rest, out_2) = write\ betw$ 

Now,  $read_1 \perp = \perp$  so  $out_1 = \perp$  and therefore

$$rw_1 \perp = (\perp, \perp)$$
  
 $rw_1 \mid \rightarrow \mid$ 

which shows that writing does not precede reading in this case.

The moral of this example is that our traces should contain some information about the point at which termination of output takes place (if indeed it does). Given a trace tr, output can only terminate after all the w actions have taken place, but it can happen at any point after that. We use the symbol  $\sqrt{\phantom{a}}$  to indicate the fact that output has (just) terminated. Intuitively, our traces for  $read_1$  and  $read_2$  will be

$$\{[\ ], [r\ a\,,\sqrt]\}$$
  
 $\{[\ ], [\sqrt], [\sqrt, r\ a]\}$ 

respectively, showing how the two processes differ. We now show how the ticks are added to the weak traces, and also introduce the notion of a terminal trace.

We now make the definition of a pretrace, on which the definition of trace is based. First we establish some notation. We write

$$finsq \perp outsq$$

when  $f(insq ++ \bot) = (\bot, outsq)$ . This is the situation in which  $f(insq) \mapsto outsq$  and the output has terminated. This is signalled by the fact that the output sequence is definite and not terminated by  $\bot$ . Now, we also say that a sequence is completed if  $\sqrt{}$  is a member of it — the choice of terminology should be obvious, as the presence of a tick is intended to mark the point at which output termination takes place. Now, because of this, we say

#### Definition

A sequence tr is a pretrace of f if and only if

- 1. tr is a weak pretrace,
- 2. If tr is completed then f  $tr [ in \downarrow tr [ out and$

$$f tr' [in \downarrow tr' [out]$$

for all completed complete initial segments tr' of tr.

3. For all initial segments tr' of tr, if f  $tr' \lceil in \downarrow tr' \rceil$  out then tr' is either a completed initial segment of tr, or is immediately followed by  $\sqrt{\ in\ tr}$ .

#### Definition

A sequence tr is a trace of f if and only if it is an initial segment of a pretrace of f.

The preceding definitions succeed in capturing sufficient information about the termination of the output of a process, as we shall see from the examples in the next section. There is another phenomenon dual to output termination — can we find out the point after which the remainder of the input can be passed to a succeeding process? This information is contained in the terminal traces, which we define now:

## Definition

A sequence tr is a  $terminal\ trace$  of f if and only if

- 1. tr is a completed pretrace of f,
- 2. For all lists x,

$$f((tr \mid in) ++ x) = (x, (tr \mid out))$$

The second condition means that whatever follows the input portion of tr in the input stream is actually passed to a succeeding process. This has the consequence that

$$ftr [in \downarrow tr [out$$

but need not be equivalent to this. Consider the examples

$$null x = (x, [])$$

$$coy x = (\bot, [])$$

Each has as collection of traces

 $\{[\sqrt{}]\}$ 

since

$$null \perp = (\perp, [])$$

$$coy \perp = (\perp, [])$$

The null process passes the remainder of the input stream, so has terminal trace  $[\sqrt{\ }]$ , whereas coy will not ask for more input yet will not pass the remainder.  $[\sqrt{\ }]$  is a terminal trace of null, but not of coy.

In the following section we re-examine the examples we looked at in section 13.

# 16 The examples revisited — traces and terminal traces

In this section we re-examine the examples from section 13, and look at their traces. Two points emerge during the discussion.

- We show that the *weak* pretraces of a process are insufficient to characterise the behaviour of a process in context by supplying two examples, 5 and 6, with the same weak pretraces but with different behaviours in context.
- The √ symbol can only appear in a trace after all the *write* items. Clearly it would be counter-intuitive for this to fail, but we exhibit a proof of the fact in example 3 below.

## Example 1

$$write in = (in, [message])$$

We want to find the sequences insq and outsq such that

$$write insq \downarrow outsq$$

write 
$$x = (\bot, y)$$

if and only if  $x = \bot$  and y = [message], so

$$write [] \downarrow [message]$$

[ $w \ message$ ] is a possible pretrace — the only problem with it is that it does not contain acknowledgement of the closing of the output. To do this we append a tick. [ $w \ message$ ,  $\sqrt{}$ ] is now a pretrace, and so the set of traces will be

$$\{[], [w \ message], [w \ message, \sqrt]\}$$

Are there any terminal traces? If we refer back to the defining equation, we can see that

$$write([]++in)=(in,[message])$$

so that [w message] will indeed be one. It is not hard to see that it is the only one.

# Example 2

$$read(a:x) = (x,[])$$

We saw that the weak pretraces were

$$\{[\ ], [r\ a]\}$$

and so, since the  $\sqrt{}$  must follow all writes, we have two possible pretraces:  $[\sqrt{}, r \ a]$  and  $[r \ a, \sqrt{}]$  — we can see that completion does take place at some point, since the output is  $[\ ]$ . For the first possibility, we need to show that

$$read[]\downarrow[]$$

since  $[\sqrt{\ }]$  is a completed initial segment of  $[\sqrt{\ },r\ a]$ . Looking back at section 13 we see that this is not the case. What about the second candidate? The only completed initial segment is the sequence itself, and we have that

$$read[a] \downarrow []$$

assuring us of the pretrace property. In fact,

$$read([a] ++ x) = (x, [])$$

so that this pretrace will be terminal too.

### Example 3

$$echo(a:x) = (x, [a])$$

echo has [] and [ $r \ a, w \ a$ ] as its weak pretraces. What are its potential pretraces? We claimed that a tick could only follow all the writes in a pretrace — intuitively this is clear, but can we see a formal reason for it? If  $\sqrt{}$  precedes the writing of some item, b say, then there are two sequences  $outsq_1$  and  $outsq_2$ , the first a proper initial segment of the other, which will contain b, with corresponding input sequences  $insq_1 \subseteq insq_2$  and

$$f(insq_i ++ \bot) = (\bot, outsq_i)$$

Since  $insq_1 \sqsubseteq insq_2$  we should have by monotonicity  $outsq_1 \sqsubseteq outsq_2$  but this is not the case, as  $outsq_2$  is a proper extension of  $outsq_1$ , a contradiction.

This means that there is only one possible completed pretrace,  $[r \ a, w \ a, \sqrt]$ , and this will indeed be one, since

$$echo[a] + + \bot = (\bot, [a])$$

that is

$$echo\ [a]\downarrow [a]$$

Also we have

$$echo[a] ++ x = (x, [a])$$

for every x, making the pretrace terminal. As the complete initial segment  $[\ ]$  is not completed, the fact that it is a weak pretrace makes it a pretrace immediately.

## Example 4

$$promptin_1 (a:x) = (x, [prompt])$$

A similar analysis to the above leads us to conclude that the traces take the form

$$\{[\ ],[r\ a],[r\ a,w\ prompt,\sqrt]\}$$

with the final trace being terminal.

## Example 5

$$promptin_2 \ y = (tl \ y, [prompt])$$

The set of weak traces is

$$\{[\ ], [w\ prompt], [w\ prompt, r\ a]\}$$

so there are two possible positions for the  $\sqrt{\ }$  in the sequence  $[w\ prompt, r\ a]$ , either before or after the final read. Now,

$$promptin_2[] \downarrow [prompt]$$
  
 $promptin_2[a] \downarrow [prompt]$ 

The first of these implies that the  $\sqrt{}$  must appear immediately after the writing of the prompt, as otherwise we contradict clause 3 of the definition of a pretrace. Our set of traces will be

Observe that

$$promptin_2[a] ++ x = (x, [prompt])$$

for every x, so that the final trace will be terminal.

## Example 6

$$\begin{array}{lll} promptin_3 \; x & = & \left(y, [prompt] + + \; rest\right) \\ & & where \\ & \left(y, rest\right) = \left(tl \; x, [\;]\right) \; , \; \; existstail \; x \\ & existstail \; (b:z) = True \end{array}$$

This provides a variant of the previous two functions.

$$promptin_3 \ x \neq \bot$$

for all x.

$$promptin_3 x ++ \bot = (\bot, z)$$

if and only if either  $x=[\,]$  and  $z=[prompt]++\perp$  or x=[a] and z=[prompt]. This means that

$$promptin_3 [] \mapsto [prompt]$$

without termination, and

$$promptin_3 [a] \downarrow [prompt]$$
 (\*)

Clearly  $promptin_2$  and  $promptin_3$  have the same weak pretraces because of these relations, but their pretraces will be different. In particular,  $[w \ prompt, \sqrt]$  is not a pretrace, since that would require that

$$promptin_3 [] \downarrow [prompt]$$

which we said was not the case. Now, the only possible candidate pretrace is

$$[w\ prompt, r\ a, \sqrt]$$

It has a single completed initial segment which by (\*) has the right property, making this a pretrace (because  $[w \ prompt]$  is not a completed initial segment we only require its weak pretrace property).

The last two examples serve to emphasise the fact that weak traces and pretraces are insufficient to characterise the behaviour of processes in context.

# 17 The definition of sequential composition

This section introduces the operation of sequential composition on trace sets and proceeds to prove that the sq function (defined in section 9 and appearing below) which combines interactions is sound with respect to this operation. We shall see that the operation has some subtlety, and will show why we needed to introduce the machinery of ticks and terminal traces.

We saw in section 15 that processes complete their behaviour in two different ways

- The output is completed a definite list is returned as the output portion
  of the result.
- The remainder of the input is passed to a succeeding process.

Recall also the discussion in section 10, in which we explained how evaluation was driven by the need to print — in other words evaluation is  $demand\ driven$ . How does this affect the sequential composition of two processes using the sq function?

$$sq \ f_1 \ f_2 \ in = (rest, out_1 ++ out_2)$$
 
$$where$$
 
$$(betw, out_1) = f_1 \ in$$
 
$$(rest, out_2) = f_2 \ betw$$

The output produced by the function is  $out_1 ++ out_2$ . The printer will demand the item-by-item evaluation of  $out_1$  to be followed on its completion by a similar evaluation of  $out_2$ . Note that  $out_2$  is only examined if and when  $out_1$  is completed. Once this happens, we begin to take output from the second process.

What does this have to suggest about the traces of the process

$$sq f_1 f_2$$
?

Clearly any trace of  $f_1$  should be a trace of the composite, and we would expect that if (and only if) a trace  $tr_1$  of  $f_1$  is completed, its effect can be followed by that of described by a trace  $tr_2$  of  $f_2$ . In other words, we might expect that  $tr = tr_1 + tr_2$  would be a trace of the composite. This is roughly right, except for two provisos.

- 1. As we remarked in section 10, writing is eager, and a write at the start of  $tr_2$  might overtake a read at the end of  $tr_1$ . This will indeed happen after the output of  $f_1$  is completed, so any read after the tick  $(\sqrt{})$  will be overtaken by a write from the front of  $tr_2$ .
- 2. We have not demanded that the trace  $tr_1$  be terminal, *i.e.* that  $f_1$  pass the remainder of its input to  $f_2$ . In case  $tr_1$  is not terminal, none of the reads in  $tr_2$  (nor any of the writes which follow such a read) can be performed. Writes at the front of the sequence can and will be performed (before any reads following the tick, as in 1.). On the other hand, if a trace  $tr_1$  is terminal, then all the following actions can be performed.

We can now formulate our definition.

Suppose that  $t_1$  is a completed trace and  $t_2$  is a pretrace.  $t_1$  will take the form

$$t_1 = t_{11} + + [\sqrt{]} + + t_{12}$$

where  $t_{12}$  will consist exclusively of reads (by the monotonicity argument we gave in example 3 of section 16 above).  $t_2$  will take the form

$$t_2 = t_{21} + + t_{22}$$

where  $t_{21}$  consists entirely of writes, and  $t_{22}$  begins either with a read or a  $\sqrt{}$ . We define the sum of  $t_1$  and  $t_2$ ,

$$t_1 \oplus t_2$$

to be

$$t_{11} ++ t_{21} ++ t_{12} ++ t_{22}$$

except in the case that  $t_{22} = [\sqrt{] + t_{22}}'$  when it is

$$t_{11} + t_{21} + t_{12} + t_{12} + t_{22}'$$

To explain the definition in words — the reads following the tick in  $t_1$  are overtaken by the writes which begin  $t_2$ . Termination of output happens at the

same point, unless that point is immediately after the initial block of writes which overtake  $t_{12}$ . In this case, termination will take place immediately after the block of writes, and before  $t_{12}$ . Note that if either  $t_{12}$  or  $t_{21}$  is empty, that is if  $t_1$  ends with a tick (which will certainly happen if its last action is to write) or if  $t_2$  begins with a read, then

$$t_1 \oplus t_2 = t_1 ++ t_2$$

We also define the partial sum of  $t_1$  and  $t_2$ ,

$$t_1 \otimes t_2$$

to be

$$t_{11} ++ t_{21} ++ t_{12}$$

except in the case that  $t_{22} = [\sqrt{] + + t_{22}'}$  when it is

$$t_{11} + t_{21} + t_{12} + t_{12}$$

Again, to explain the definition, this is a case where the reads at the end of  $t_1$  are overtaken by the writes at the front of  $t_2$  — the rest of  $t_1$  is lost, however. This combinator models the combination of traces when  $t_1$  is not terminal, the case we discussed in point (2.) above.

#### Definition

We define the sum  $S_1 \oplus S_2$  of sets of pretraces  $S_1$ ,  $S_2$  to consist of

- 1. The non-completed members of  $S_1$ ,  $tr_1$ , say.
- 2. The sums  $tr = tr_1 \oplus tr_2$  of terminal traces  $tr_1$  from  $S_1$  with members  $tr_2$  of  $S_2$ .
- 3. The partial sums  $tr = tr_1 \otimes tr_2$  of non-terminal completed traces  $tr_1$  from  $S_1$  with members  $tr_2$  of  $S_2$

We now look at the proof that this sum embodies "lazy" sequential combination of processes.

# 18 Proving the correctness of sequential composition

In this section we prove that the sq function implements sequential composition, as defined in the previous section. This has the formal statement:

**Theorem 2** If  $S_i$  is the set of pretraces of  $f_i$  then  $S_1 \oplus S_2$  is the set of pretraces of  $sq\ f_1\ f_2$ .

#### Proof:

We prove the result in two parts. First we show that every member of  $S_1 \oplus S_2$  is a pretrace of the composition of  $f_1$  and  $f_2$ , and then show that each such pretrace is a member of the sum set.

Members of  $S_1 \oplus S_2$  take three forms, as outlined in the definition. We look at these cases in turn, starting with

#### Case 1:

This is the case of a non-completed pretrace of  $f_1$ , tr, say. Since this is not completed,

$$f_1 (tr [in ++ \bot) = (\bot, tr [out ++ \bot)$$
 (†)

or, in the case that  $tr \mid out = \bot$ , we might have

$$f_1(tr \mid in ++ \perp) = \perp \quad (\ddagger)$$

Consider the form of f applied to the same argument,

$$f\left(tr \mid in++\perp\right) = (rest, out_1 ++ out_2)$$

$$where$$

$$(betw, out_1) = f_1 (tr \mid in++\perp)$$

$$(rest, out_2) = f_2 betw$$

In either case,  $betw = \bot$ , so clearly  $rest = \bot$  — here we are making the assumption that for all our interactive functions, g, and all input streams x, if

$$g x = (rest, out)$$

then rest is a final segment of x, or is  $\perp$ , which can be considered to be the degenerate final segment. Now, in case  $(\frac{1}{1})$ 

$$f(tr [in ++ \bot) = (\bot, tr [out ++ \bot)$$

and in case (‡)

$$f_1 (tr [ in ++ \bot) = (\bot, \bot)$$

so that in both cases,

$$f tr [in \mapsto tr [out]$$

Similar arguments apply to any complete initial segment of tr, which means that tr is a pretrace of the function f.

#### Case 2:

is the case in which we take  $tr = tr_1 \oplus tr_2$ , where  $tr_1$  is a (completed) terminal trace of  $f_1$ , and  $tr_2$  is a pretrace of  $f_2$ . Note that

$$tr \lceil in = (tr_1 \oplus tr_2) \lceil in$$
  
=  $tr_1 \lceil in + tr_2 \rceil in$ 

$$tr \mid out = (tr_1 \oplus tr_2) \mid out$$
  
=  $tr_1 \mid out ++ tr_2 \mid out$ 

Now,

$$f\left(tr \mid in + + \perp\right) = (rest, out_1 ++ out_2)$$

$$where$$

$$(betw, out_1) = f_1 (tr \mid in ++ \perp)$$

$$(rest, out_2) = f_2 betw$$

Since  $tr_1$  is a terminal trace of  $f_1$ ,

$$f_1(tr \lceil in) ++ \perp = f_1(tr_1 \lceil in ++ tr_2 \lceil in ++ \perp)$$
$$= (tr_2 \lceil in ++ \perp .tr_1 \lceil out)$$

so  $betw = tr_2 [in ++ \bot and out_1 = tr_1 [out. This means that]$ 

$$\begin{array}{cccccc} f_2 \; betw & = & f_2 \; (tr_2 \; \lceil \; in \; + + \; \bot) \\ & = & (\bot, tr_2 \; \lceil \; out) & (\dagger) \\ & or & (\bot, tr_2 \; \lceil \; out \; + + \; \bot) & (\ddagger) \\ & or & \bot & (\dagger\dagger) \end{array}$$

In these cases we have, respectively,

$$f(tr \lceil in ++ \bot) = (\bot, tr_1 \lceil out ++ tr_2 \lceil out) \qquad (\dagger)'$$

$$or (\bot, tr_1 \lceil out ++ tr_2 \lceil out ++ \bot) \qquad (\dagger)'$$

$$or (\bot, tr_1 \lceil out ++ \bot) \qquad (\dagger\dagger)'$$

which implies that

$$f tr [in \mapsto tr [out]$$

We now have to check that

$$f tr' [in \mapsto tr' [out]$$

for every complete initial segment tr' of tr. What form do these complete initial segments take? Our proof follows a case analysis according to the form of tr. There are two cases.

Case 2A:  $tr = t_{11} + t_{21} + t_{12} + t_{22}$ 

Case 2B: 
$$tr = t_{11} + t_{21} + t_{21} + t_{12} + t_{12} + t_{22}$$
, where  $t_{22} = [\sqrt{t_{12}} + t_{22}]$ 

If we first look at the input/output properties of f, that is, those properties embodied by the ' $\mapsto$ ' relation, we can treat the cases 2A and 2B together. It

is only when we look at completed initial segments of the two that we have to draw a distinction, and it is only for these cases that the position of the  $\sqrt{}$  is important. (Note that we call the properties of ' $\mapsto$ ' the input/output properties since for a given sequence tr whether f  $tr \lceil in \mapsto tr \lceil out$  simply depends on the input/output actions as summarised in the sequences  $tr \lceil in$  and  $tr \lceil out$ .)

We are to examine the complete initial segments of tr. (To be specific we look at the case 2A, but the two cases are exactly the same, structurally.) These will be

- (I) complete initial segments of  $t_{11}$ .
- (II)  $t_{11} + t_{21} + t'$  where t' is a complete initial segment of  $t_{12}$  or  $t_{12}$  itself.
- (III)  $t_{11} + t_{21} + t_{12} + t'$  where t' is a complete initial segment of  $t_{22}$ .

We examine these sub-cases in turn.

#### Sub-case (I):

 $t' \subset_c t_{11}$ , so we know that

$$f t' [in \mapsto t' [out]$$

and moreover since t' is a proper initial segment of  $t_{11}$ , and therefore is a proper initial segment of the shortest completed segment,

$$f_1 (t' \upharpoonright in ++ \bot) = (\bot, t' \upharpoonright out ++ \bot)$$
  
 $or \bot$ 

(the latter case can only happen if t' [ out is [ ] ). Now,

$$f(t' \mid in ++ \perp) = (rest, out_1 ++ out_2)$$

where rest is a final segment of  $\bot$ , and  $out_1$  is terminated by  $\bot$ , so we have in the two cases

$$= (\bot, t' \upharpoonright out ++ \bot)$$
$$= (\bot, \bot)$$

which in either case gives

$$f(t' [in) \mapsto (t' [out)$$

as required.

#### Sub-case (II):

We are looking at  $t_{11} + t_{21} + t'$  where t' is a complete initial segment of  $t_{12}$ , or is  $t_{12}$  itself. Now, t' is a complete initial segment of  $t_{12}$  if and only if

$$\tilde{t} = t_{11} + + [\sqrt{1 + t'}]$$

is a complete initial segment of  $t_1$  or is  $t_1$  itself. Because of the position of  $\sqrt{\ }$  in  $\tilde{t}$ .

$$f_1$$
  $(\tilde{t} \lceil in) \downarrow (\tilde{t} \lceil out) \quad (\dagger)$ 

Since  $t_{21}$  consists entirely of writes, and  $t_{12}$ , t' of reads,

$$(t_{11} + t_{21}) \mid in = t_{11} \mid in$$

$$(t_{11} + t_{21} + t') \mid out = (t_{11} + t_{21}) \mid out$$

$$= (t_{11} + t_{12} + t_{21}) \mid out$$

$$= (t_{1} + t_{21}) \mid out$$

Now, by (†),

$$f_{1} (t_{11} ++ t_{21} ++ t') \lceil in ++ \bot = f_{1} (t_{11} ++ t') \lceil in ++ \bot$$

$$= (\bot, (t_{11} ++ t') \lceil out)$$

$$= (\bot, t_{11} \lceil out) \quad (\ddagger)$$

So, looking at the results of f,

$$f(t_{11} + + t_{21} + + t') [in + + \bot]$$

By (‡),

$$f_1\ (t_{11}++\ t_{21}++\ t')\ \lceil\ in++\ \bot=(\bot,t_{11}\ \lceil\ out)$$

so  $betw = \bot$ , giving  $rest = \bot$ . Since  $t_{21}$  is a complete initial segment of  $t_2$ ,

$$f_2 [] \mapsto t_{21}$$

so,

$$f(t_{11} ++ t_{21} ++ t') \cap in \mapsto t_{11} \cap out ++ t_{21}$$

$$= (t_{11} ++ t_{21}) \cap out$$

$$= (t_{11} ++ t_{21} ++ t') \cap out$$

giving the required input/output property. Now we consider the third case

#### Sub-case (III):

Here we consider the complete initial segment

$$t_{11} ++ t_{21} ++ t_{12} ++ t'$$

where t' is a complete initial segment of  $t_{22}$ . Note that  $t_{21} ++ t'$  is a complete initial segment of  $t_2$ , so

$$f_2(t_{21} ++ t') [in \mapsto (t_{21} ++ t') [out (\dagger)]$$

Looking at the value of f,

$$f(t_{11} + + t_{21} + + t_{12} + + t') [in + + \bot]$$

= 
$$(rest, out_1 ++ out_2)$$
  
 $where$   
 $(betw, out_1) = f_1 (t_{11} ++ t_{21} ++ t_{12} ++ t') [in ++ \bot$   
 $(rest, out_2) = f_2 betw$ 

Note that  $t_1$  is a terminal trace of  $f_1$ , so

$$f_1(t_1 ++ t_{21} ++ t') [in ++ \bot = ((t_{21} ++ t') [in ++ \bot, t_1 [out)]$$

also,

$$(t_{11} + t_{21} + t_{12} + t')$$
  $[in = (t_1 + t_{21} + t')$   $[in = (t_1 + t_{21} + t')$ 

so

$$betw = (t_{21} ++ t') \mid in ++ \perp$$
$$out_1 = t_1 \mid out$$

Now, by (†) we can deduce that  $out_2$  is one of  $(t_{21} + + t') \cap out$  and  $(t_{21} + t') \cap out + + \perp$ , so

$$f(t_{11} + t_{21} + t_{12} + t') [in \mapsto t_1 [out + (t_{21} + t') [out + (t_{11} + t_{12} + t') ]out$$

$$= (t_{11} + t_{21} + t_{12} + t') [out$$

This establishes the input/output property for the third and final type of complete initial segment.

What stage have we reached in our proof? We are mid-way through proving case 2, that  $t_1 \oplus t_2$  is a pretrace. We have checked the input/output property of the complete initial segments of  $t_1 \oplus t_2$ , so we should now check the *convergence* or *completion* properties of the sequence. This amounts to checking whether or not a ' $\checkmark$ ' should be included in the sequence, and if so, where. To give a full proof we should check all the cases, but intuitively it should be clear that our definition of  $\oplus$  places a  $\checkmark$  in the right position — we just make a few remarks to reinforce that intuition.

Output of the composite is complete if and only if output of both the components is completed; the completion of the first is a pre-requisite for the output of the second to be produced — the completion of this completes the composite.

This completion will occur at the same place (within  $t_{22}$ ) unless it occurs at the start of  $t_{22}$ . In general, the writes in  $t_{21}$  overtake the reads in  $t_{12}$ , and in the special case the completion of output can overtake these reads too — this is why we move  $\sqrt{}$  forward in this and only this case.

We now look at the third and final case (3.), that of the partial sum  $t_1 \otimes t_2$ , where  $t_1$  is a non-completed pretrace of  $f_1$ , and  $t_2$  a pretrace of  $f_2$ .  $t_1 \otimes t_2$  takes one of the forms

$$t_{11} ++ t_{21} ++ t_{12}$$
 $t_{11} ++ t_{21} ++ [\sqrt{]} ++ t_{12}$ 

(the latter case when  $t_{22} = [\sqrt{]} + + t'_{22}$ ) This case follows exactly the treatment of case (II) above, where we assumed that t' was a complete initial segment of  $t_{12}$  or  $t_{12}$  itself.

This completes the proofs of the three cases, and thus the first half of the proof — each member of the sum set is a pretrace of the composition of the functions.

We now aim to show the converse, that every pretrace of the composite is a member of the sum set. In this part of the proof we shall rely on the determinacy theorem as seen in section 14. This theorem can be extended to pretraces proper, rather than weak pretraces — we leave this extension as an exercise for the reader. We now proceed by examining the general form of the definition of f. Consider sequences insq, outsq so that

$$f insq \mapsto outsq$$

We have

$$f insq ++ \perp = (\bot, out_1 ++ out_2)$$

$$where$$

$$(betw, out_1) = f_1 insq ++ \perp \qquad (\dagger)$$

$$(\bot, out_2) = f_2 betw$$

What possibilities are there for  $(\dagger)$ ? If  $out_1$  is terminated by  $\bot$ , we have

$$out_1 ++ out_2 = out_1$$

and so

$$out_1 = outsq ++ \bot$$

betw is a final segment of

$$insq ++ \bot$$

since the output from  $f_1$  is not completed, we assume that *betw* will be  $\bot$ —the input will not be passed until the output is completed—we examine this assumption further in the next section (19). Given *betw* is  $\bot$  we have

$$f_1 insq ++ \perp = (\perp, outsq ++ \perp)$$

The determinacy theorem states that there is a unique pretrace tr of  $f_1$  so that tr [in = insq and tr [out = outsq. Note that this pretrace is not completed.]

Now consider the case that  $out_1$  is definite, that is not terminated by  $\bot$ . There are two sub cases.

#### Sub-case 1:

$$f_1 insq ++ \perp = (\perp, out_1)$$

so  $betw = \bot$ , and therefore

$$f_2 \ betw = (\bot, out_2)$$

By the determinacy theorem there is a unique pretrace tr of  $f_1$  such that  $tr \lceil in = insq$  and  $tr \lceil out = outsq$ .  $out_2$  will be a pretrace of  $f_2$  and the appropriate interleaving of insq and outsq will be provided by  $tr \otimes out_2$ . Finally,

#### Sub-case 2:

$$f_1 insq ++ \perp = (betsq ++ \perp, out_1)$$
  
 $f_2 betsq ++ \perp = (\perp, out_2)$ 

Arguing as in case 1, we can find pretraces  $tr_1$  (terminal, in fact) and  $tr_2$  of the two functions so that  $tr_1 \oplus tr_2$  is an interleaving of the sequences insq and outsq with the pretrace property.  $\square$ 

This concludes our proof that sequential composition, as defined by the operation  $\oplus$  on trace sets, is properly implemented by the function sq. Note that we made two assumptions during the course of the proof.

1. For all interactive functions, g, and all input streams x, if

$$g \ x = (rest, out)$$

then rest is a final segment of x, or is  $\bot$ , which can be considered to be the degenerate final segment.

2. If insq, outsq are sequences, then if

$$f insq ++ \perp = (betw, outsq ++ \perp)$$

then  $betw = \bot$ . This is discussed further in the next section.

# 19 A Black Hole

What does it mean for there to be a non-empty sequence betsq such that

$$f insg ++ \perp = (betsg ++ \perp, outsg ++ \perp)$$
?

By the monotonicity of the function f, there is some initial segment insq' of insq such that

$$f insg' ++ x = (x, outsg ++ \bot)$$

for all x. This means that we have a pretrace (interleaving insq' and outsq), tr say, which is not completed — its output is not completed — yet it passes its input. Because of this latter property (using monotonicity) there is no chance that by reading further input the output can be completed. Such a process 'locks' in a particularly awkward way; no output from succeeding processes will ever be demanded by the printing mechanism (c.f. section 11). All that we can do is type ahead an arbitrarily great amount of input, which will never be read nor rejected.

For the reasons stated above, we have chosen to exclude such processes from our analysis. However, we can check for such behaviour, by looking for non-completed "pseudo-terminal" traces, such as those above, and make the appropriate modifications to our definitions to recognise the 'type ahead'. We trust the interested reader to do this for him- or herself.

# 20 Conclusions

We have explored the general behaviour of our streams-as-lazy-lists model for interactions, and have shown how the behaviour of sequential composition can be explained in terms of trace sets. If we ensure that output termination, as signalled by  $\sqrt{\ }$ , occurs only at the end of terminal traces, we can be sure that no overtaking takes place, and that composition of traces is simply concatenation. On the other hand, we have the formal means by which we can explain more exotic interactions. As can be seem from the first part of the paper, sq is the crucial combinator as together with a simple choice combinator, primitive reading and writing operations and recursion we are able to define all the other higher-level combinators.

#### Definition

We call a process, f say, read-strict if for no trace of f does  $\sqrt{}$  precede an input item (and therefore  $\sqrt{}$  precedes no other items).

It is not difficult to show that the basic operations of section 5 are read-strict and that the combining forms introduced there preserve read-strictness; the only non-trivial case is that of sq, and that result follows from our analysis of sq above. Now, if we take the sequential composition of two read-strict processes we can see that no "overtaking" of reads by writes can take place (again, by our analysis of sq), and so we would claim that the interleaving behaviour of the functions of section 5 is the predictable as we suggested. On the other hand, as we saw from the examples in 6, it is all too easy to go astray if we adopt an adhoc approach.

Another approach to I/O in a functional language is suggested by Backus, Williams and Wimmers in [1] and followed up in [7]. In the latter paper they suggest that the lazy stream, or 'pure', approach which we adopt here is one which is less convenient to use than one in which every function is taken to have a side-effect on the history of the I/O devices. We would prefer to see our approach as complementary to theirs. Indeed, as we explained in 4, we can see our type

$$interact * **$$

as a type of functions with side-effects on I/O.

As Williams and Wimmers remark, many functions fail to affect the history. This phenomenon is manifested here by the apply operation,

$$apply :: (* \rightarrow **) \rightarrow interact * **$$

which turns a "true" function into one with (trivial) I/O side-effects. Once we realise this we can see that properties of many functions will carry over, just as Williams and Wimmers suggest.

The major advantage that we see in our approach is that we have a *purely* functional model of I/O, and so one to which we can apply the accepted methods of reasoning. Again, as we rearked in the conclusion to the first part of the paper we see no need to combine the functional one with any other in order to perform interactive I/O.

I am grateful to my colleagues at the University of Kent for various discussions about interactions and processes, and for using the interaction combinators supplied in the earlier paper and giving me valuable feedback about their behaviour.

## References

- [1] John Backus, John H. Williams, and Edward L. Wimmers. FL language manual (preliminary version). Technical Report RJ 5339 (54809) 11/7/86, IBM Research Division, 1986.
- [2] Robert Cartwright and James Donahue. The semantics of lazy (and industrious) evaluation. Technical Report CSL-83-9, Xerox, Palo Alto Research Center, 1984.
- [3] C. A. R. Hoare. Communicating Sequential Processes. Prentice-Hall International, 1985.
- [4] A. Pnueli. Applications of temporal logic to the specification and verification of reactive systems: a survey of current trends. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, Current Trends in Concurrency. Overviews and Tutorials, number 224 in Lecture Notes in Computer Science. Springer Verlag, 1986.

- [5] Simon J. Thompson. A logic for Miranda. Draft of a paper describing a logic in which to reason about Miranda programs. Based on PP $\Lambda$ ., March 1987.
- [6] David A. Turner. Miranda: a non-strict functional language with polymorphic types. In J. P. Jouannaud, editor, Functional Programming Languages and Computer Architecture. Springer-Verlag, 1985.
- [7] John H. Williams and Edward L. Wimmers. Sacrificing simplicity for convenience: Where do you draw the line? In *Proceedings of POPL*, 1987.