Interactive functional programs*

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

October 3, 1989

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 an expanded version of the paper [1] 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[1] [2].

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

*An expanded version of this paper will appear in *Research Topics in Functional Programming*, David Turner (ed.), Addison-Wesley, 1990

[1]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.

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.

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 $\perp$ for the undefined list (indeed we write it for the undefined object of any type). Using this notation we see that

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

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.
- On the other hand, given an item b, the stream

  $$b : y$$

  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

$$\begin{array}{rcll} nums'\ n\ m & = & [\,] & , n > m \\ & = & n\ :\ nums'\ (n+1)\ m & , otherwise \end{array}$$

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.

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:

$$\begin{array}{rcll} double{-}space\ (a : x) & = & a : double{-}space\ x & , a \sim= newline \\ & = & a : a : double{-}space\ x & , otherwise \\ double{-}space\ [\,] & = & [\,] & \end{array}$$

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 *st*and*d*ard *in*put), 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 *echo*ing 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−on*, starts off in double spacing mode.

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

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.

$$line{-}len :: input \rightarrow (input, output)$$

```
line-len in
  =  (rest, out)
     where
     out = show (#line)
     (line, rest) = get-line [ ] in
     get-line front (a : x)
       = get-line (front ++ [a]) x    , a ~= newline
        = (front, x)                  , otherwise
```

get−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−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−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−len* so that it will print an accumulated total number of characters after each line, as well as the length of the line itself.

$$line{-}len{-}deluxe :: interact\ num\ num$$

$$\begin{array}{ll} \multicolumn{2}{l}{line{-}len{-}deluxe\ (in, tot)} \\ = & (rest, newtot, out) \\ & where \\ & (line, rest) \qquad\qquad = get{-}line\ [\,]\ in \\ & len \qquad\qquad\qquad\quad = \#\ line \\ & newtot \qquad\qquad\quad = tot + len \\ & out \qquad\qquad\qquad\quad = show\ len ++ show\ newtot \end{array}$$

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

$$get{-}char :: interact\ *\ char$$

$$get{-}char\ ((a : x), st) = (x, a, [\,])$$

and to write single characters,

$$put{-}char :: char \rightarrow interact * *$$

$$put{-}char\ ch\ (in, st) = (in, st, [ch])$$

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

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

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

$$(rest, final, out{-}second)$$

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,

$$out{-}first\ ++\ out{-}second$$

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

$$cond\ * \rightarrow interact\ *\ ** \rightarrow interact\ *\ ** \rightarrow interact\ *\ **$$

$$\begin{array}{lcll} alt\ condit\ inter1\ inter2\ (in, st) & = & inter1\ (in, st) & , condit\ st \\ & = & inter2\ (in, st) & , otherwise \end{array}$$

$cond\ * == * \rightarrow bool$ 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

$$\begin{aligned} & while\ condit\ inter \\ = \quad & loop \\ & where \\ & loop = alt\ condit\ (inter\ \$sq\ loop)\ skip \end{aligned}$$

'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:

$$repeat :: cond * \rightarrow interact\ *\ * \rightarrow interact\ *\ *$$

$$\begin{aligned} & repeat\ condit\ inter \\ = \quad & inter\ \$sq\ (while\ not{-}condit\ inter) \\ & where \\ & not{-}condit = (\sim).condit \end{aligned}$$

$\sim$ is the boolean negation function, so that *not−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

$$\begin{aligned} rep{-}inter \quad &:: \quad interact\ num\ num \\ rep{-}inter \quad &= \quad repeat\ sufficient\ line{-}len{-}deluxe \end{aligned}$$

we can define our full interaction,

$$full{-}inter :: input \rightarrow output$$

by

$$\begin{aligned} & full{-}inter\ in \\ = \quad & out \\ & where \\ & (rest, final, out) = rep{-}inter\ (in, 0) \end{aligned}$$

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:

$$try\ (a : x) = \text{"}Prompt : \text{"} \mathbin{++} [a]$$

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

$$again\ x = \text{"}Prompt : \text{"} \mathbin{++} [hd\ x]$$

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 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.

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] Simon J. Thompson. *Interactive functional programs.* Technical Report 48, Computing Laboratory, University of Kent at Canterbury, 1987. An extended version of this paper, containing further discussion of the examples, and full proofs of the theorems. A slightly amended version of the paper will appear in *Research Topics in Functional Programming*, David Turner (ed.), Addison-Wesley, 1990.

[2] 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.