An alternative approach to I/O

Maarten Fokkinga and Jan Kuper
Dept. of Computer Science, University of Twente, Netherlands
{fokkinga,jankuper}@cs.utwente.nl

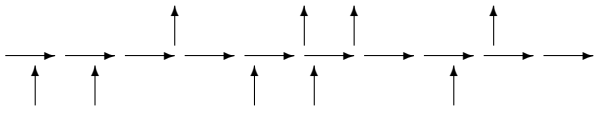
Abstract. We propose a form of input and output for functional languages that is in a sense *orthogonal* to the actual computation: certain input and output directives can be added to a completed, fully working program text, and they do neither disturb referential transparency nor necessitate to change the types of the program text. The input and output directives change the order of evaluation as little as possible (lazy evaluation remains lazy), though there is sufficient control over the order in which the input and output actions occur to make it acceptable for the user. The basic idea is that a value which is written out explicitly in the program text by way of typical example, is replaced by a special constant that asks the user to type in parts of the value, as needed by the computation.

The mechanism seems suitable for a large class of so-called interactive programs.

Introduction

1 The nature of input and output. Clearly, during execution of a program the order of input and output is relevant, e.g., the first input value should not be confused with the second input value. However, in a referentially transparent language it is to a large degree irrelevant (and entirely left to the evaluator) whether one subexpression is evaluated before or after another one. As a consequence, to deal with input/output the programmer is forced to use an explicit continuation style, treating all the inputs and outputs as two long string parameters (and taking care that output does not show up too early!), or to use a monadic style in which the same is expressed more implicitly. See for example the approach by Bird [3], Thompson [5], Clean [1, 2] and the Haskell 98 Report [4]. Both styles of programming also have the phenomenon that the types of the program with input and output differ from the case where no output is provided and all input is artificially done by explicitly writing out in the program text a typical input value.

For us the ideal would be that input and output can be added to a completed, fully working input/output-less program *without changing* the types and structure of the text. In particular, this applies to programs whose primary purpose is *not* to realize some input and output, but instead to do some computation, say leading from an initial state to a final one. For such a program, we consider input and output as something *orthogonal* to the actual computation, as suggested by the following figure:



The horizontal arrows suggest the primary computational process, leading to the *result* (normal form, possibly infinite) of the program expression. The vertical arrows suggest *input* from and *output* to a user.

The *semantics* of a program is determined by the question which result is calculated given some arguments. According to the above figure, the result is the normal form (possibly infinite) of the horizontal reduction process. All output given to the user for the sake of interaction with that user, only gives information to the user about the values of certain subexpressions (for example, the user may use that information to decide what input to give next). Hence, this output is irrelevant for the semantics of the program.

In a way, the same holds for input. Clearly, input given by the user does determine the result of a program. However, the question whether certain expressions get their values because the user types them in, or whether these values are determined by a computation process, is *not* relevant for the semantics. In other words, the semantics of a function is its argument–result behavior, but the way in which the arguments get their values does not belong to the semantics. In that sense, (the actual providing of) input is irrelevant for the semantics.

Thus, contrary to standard approaches, we treat interaction with a user as *not* belonging to the primary evaluation process, *nor* to the semantics of the program. To support this idea of orthogonality, we suggest that distinct windows will be generated to accept input, and to show output. The normal evaluation window, which is standard in all functional languages, then is reserved for showing only the final result of the program. The consequence for the programmer is that (s)he should decide in advance what belongs to the final result, and what belongs to the interaction with the user.

Typical programs for which the above approach works are interactive games like hangman and tictactoe. Such programs express a state transformation, from the initial state of the game to the final one. Ideally, it should be possible for the programmer to design a program *assuming* that all input is already available (as one or two lists of strings, say). Once the program is developed, it should be possible to add interaction with a user by means of certain *IO-directives* without changing the types, the basic structure, or the semantics of the program. We have always felt it as a nuisance of existing approaches that making such 'state transformations' interactive forces the programmer to change the program in a considerable way.

To conclude, we propose a mechanism for input/output to a human user (rather than file-IO) that doesn't disturb referential transparency. Its strength is "programmer friendliness" (input/output can be added without changing the types and the structure of the program), its weakness may be its restricted applicability (we do not know whether all conceivable input/-output can be done with our mechanism). Typical of the proposal is that there is no full control over when input and output occurs, but only over the order in which it occurs. We will show how to simulate its usage in pure Haskell, by using monads.

2 Example. To illustrate the basic idea, consider the following function, that reverses its argument list and replicates each element:

```
rev :: [[Char]] -> [Char]
rev xs = if xs==[] then "." else rev (tail xs) ++ head xs ++ head xs
```

Applied to argument `["a","b","c"]` it produces `".ccbbaa"`.

In order to have its argument read in, we apply `rev` to a suitably declared special constant rather than `["a","b","c"]`. The effect of that special constant will be that the user is prompted several times to indicate whether the input list is completed or still continues, and what the list elements are. If the user says precisely three times that the list continues, with elements `"a"`, `"b"`, `"c"`, respectively, then the very same result `".ccbbaa"` is produced as with `rev ["a","b","c"]`.

Even though '`head xs`' occurs twice in the program text, the user is prompted only once for its value. Even though in the recursive computation the *first needed* list element is `last xs`, the *first input* value plays the role of `head xs`: the order of input is the same as the order in the list, i.e., earlier elements in the list will be forced to be read in before later elements in the list. For all this, no changes need be made to the body of `rev`, and in particular its type stays the same.

Prompt messages will be integrated in the declaration of these special input constants. If some output has to be shown just before the input of `head xs`, in addition to the prompt message, then we have to add an output directive to the body, still without changing the functions' type:

```
rev xs = OUT head xs: output ->
         if xs==[] then "." else rev (tail xs) ++ head xs ++ head xs
```

or even:

```
rev xs = if xs==[] then "." else
         (OUT head xs: output -> rev (tail xs)) ++ head xs ++ head xs
```

Here *output* is an arbitrary expression. The expression '`OUT` $expr_0$`:` $expr_1$ `->` *expr*' has, apart from some input/output effect, the *same* type and semantics as *expr*. Its input/output effect is as follows: *when and if* the value denoted by $expr_0$ is read in (not necessarily due to an evaluation step "within" *expr*), then the value of $expr_1$ is shown just before the input prompt. So, the output effect depends to some degree on the order of evaluation chosen by the evaluator. For example, in the following program the `OUT` expression has not been evaluated before the recursive call has been invoked and, as explained above, the user has been prompted to type in a value for `head xs`, so the output `hello` will not be shown:

```
rev xs = if xs==[] then "." else
         rev (tail xs) ++ (OUT head xs: "hello" -> head xs) ++ head xs
```

The proposal for Input

3 Input actions. We consider 'input' to be a user action that determines to some degree a value that could have been expressed in the program text itself. The precise definition depends on the type of the value, so we first give some typical examples of these:

```
data Nat = 0 | 1 | 2 | 3 | ...
data Text = "" | "a" | ... | "abc" | ...
data List a = Nil | Cons a (List a)
data Stream a = Next a (Stream a)
```

We consider `0`, etc., and `""`, etc., to be 0-ary constructors, just like `Nil`.

Now we define that an *input action* for a value consists of determining, in some way or another (typically: hitting some keys of the keyboard or selection in a menu, possibly in a

separate window) the *outermost constructor* of the value. So, to determine the first element of a stream value no real input action is needed, since its outermost constructor `Next` is already determined by the type. A natural number value is completely input by a single input action, and similar for text values. A list may be completely input by a single input action, if it is `Nil`, but in general it requires several input actions. The input of an infinite list or stream requires infinitely many input actions.

For many programs the entire input is simply a sequence $line_0$, ..., $line_{n-1}$, each line being a string. Such a sequence can easily be modeled as a value of type `Stream Text`:

`Next` $line_0$ `( ... (Next` $line_{n-1}$ `undefined)...)`

4 Example programs. With the above datatypes the definition of `rev` reads:

```
rev xs = case xs of Nil -> "."; Cons h t -> rev t ++ h ++ h
```

Anticipating the simulation in pure Haskell, we use the `case` construct as the sole means to destruct a composite value. For illustrative purposes, we might complicate the definition somewhat by defining and using auxiliary functions `ifNil`, `hd`, and `tl`:

```
ifNil xs e1 e2 = case xs of Nil -> e1; Cons h t -> e2
hd xs = case xs of Cons h t -> h
tl xs = case xs of Cons h t -> t
rev xs = ifNil xs "." (rev (tl xs) ++ hd xs ++ hd xs)
```

For simplicity we assume that `++` has the desired effect on our type `Text`. The following stream variant of the reversal function will be no surprise either:

```
frst xs = case xs of Next f r -> f
rest xs = case xs of Next f r -> r
revs xs = if (frst xs = "N") then "." else
                   revs(rest(rest xs)) ++ frst(rest xs) ++ frst(rest xs)
```

For example, consider the following stream:

```
Next "Y" (Next "a" (Next "Y" (Next "b" (Next "Y" (Next "c" (Next "N"
                                                        undefined))))))
```

With this stream as argument, `revs` will return `".ccbbaa"`

5 Syntax: input constants. To "pick up" input values typed in by a user, our proposal extends the syntax with the notion of *input constant*, which we for clarity write as an identifier prefixed with a question mark:

```
?t   :: Text
?inp :: List Text
?nss :: List (List Nat)
```

An input constant of type t is itself an expression of type t, and may be used as such without any restriction. An input constant for a list or stream will automatically generate a series of input constants, corresponding to the constituents in the list or stream. Each generated

input constant will be labelled by a "path", which indicates for what constituent of the original list or stream it is generated. The explicitly declared input constants have, by default, the "empty path" as label. So, to formally define the semantics in terms of reduction rules we first postulate, for each input constant $?x$ and each path s, the existence of an additional input constant $?x_s$ (with the same identifier x) typed in the appropriate way:

$?x_s$`:: List a` $\Rightarrow$ $?x_{s,Cons_0}$`:: a` and $?x_{s,Cons_1}$`:: List a`
$?x_s$`:: Stream a` $\Rightarrow$ $?x_{s,Next_0}$`:: a` and $?x_{s,Next_1}$`:: Stream a`

The general scheme is that if $?x_s$ has type t and t has an n-ary constructor C, then n additional input constants $?x_{s,C_0}, \ldots, ?x_{s,C_{n-1}}$ exist, with the types as given by the constructor. Constructor `Nil` and all constructors of `Nat` and `Text` are 0-ary and, hence, do not give rise to additional input constants. The additional input constants will normally not occur in a program text, but only show up during the evaluation when formulated as a rewriting process.

6 Semantics: rewrite rules. For the input constants we postulate these reduction rules:

$?x_s$	$\rightarrow$	`0` | `1` | `2` | ...	when $?x_s$: `Nat`
$?x_s$	$\rightarrow$	`""` | `"a"` | ... | `"abc"` | ...	when $?x_s$: `Text`
$?x_s$	$\rightarrow$	`Nil` | `Cons` $?x_{s,Cons_0}$ $?x_{s,Cons_1}$	when $?x_s$: `List a`
$?x_s$	$\rightarrow$	`Next` $?x_{s,Next_0}$ $?x_{s,Next_1}$	when $?x_s$: `Stream a`

The rewrite rules for `Nat`, `Text`, and `Cons` have several alternatives in their right-hand side. The choice has to be made by the user: whenever a rewrite rule with several alternatives in the right-hand side is applied, the user is prompted for the choice; for example, the user types in or selects the outermost constructor of the desired alternative. Moreover, in order that referential transparency is not violated, the evaluator has to *share* the input constants. So, if during the evaluation an input constant $?x$ has been replicated (for example because it was the argument of $\lambda\, y. \cdots y \cdots y \cdots$), then all occurrences of $?x$ have to be replaced simultaneously with the same value. Without this requirement for sharing of input constants, it might be the case that the user is prompted twice to input one and the same value – and thus might type in two different values for the same input constant.

The prompt messages shown to the user at input actions, have to be expressed within the declaration of the input constants. We elaborate this syntactic part of the proposal not in this paper, but simply give the idea with an example (using *true* for $\lambda\, x.\, True$):

```
?inp :: List Text   where
        "Nil/Cons [Cons]?":
          (λ x.  x ≠ "" ∧ head x = 'N')  -> Nil
          true                           -> Cons ("Text?": true -> ()) ()
```

Now, at each input action for `?inp` or any tail of `?inp`, namely $?\texttt{inp}_{Cons_1,\ldots,Cons_1}$, the prompt text '`Nil/Cons [Cons]?`' is shown to the user, and if the user's actual input satisfies the predicate $(\lambda\, x.\ x \neq \texttt{""} \wedge \mathit{head}\ x = \texttt{'N'})$ the first alternative, `Nil`, is chosen. Otherwise, if the actual input satisfies predicate *true* the `Cons` alternative is chosen. Similarly, if an input action for $?\texttt{inp}_{Cons_1,\ldots,Cons_1,Cons_0}$ occurs, the user is prompted with '`Text?`'.

7 Semantics: order of input actions. The semantics of the previous paragraph might give some surprises! For example, suppose we declare an input constant `?inp::List Text` and then invoke `rev ?inp`, or we declare `?inp::Stream Text` and then invoke `revs ?inp`. The following interaction may occur under lazy evaluation (showing only the choices made):

`Cons; Cons; Cons; Nil; "a"; "b"; "c"`	leading to the result:	`".aabbcc"`
`"Y"; "Y"; "Y"; "N"; "a"; "b"; "c"`	leading to the result:	`".aabbcc"`

The *last* element of the argument list has been input *first*! Clearly, what we need is some control over the order in which the input actions for the complete determination of `inp` are performed. As it stands, that order is determined by the evaluation strategy of the evaluator (lazy, strict, or whatever).

We have experimented for some time with a means for the programmer to specify a 'syntactic' partial order between subexpressions, with the intended effect that the 'semantic' order in which the input actions occur during the evaluation is not in conflict with the specified 'syntactic' partial order on the subexpressions (which carries over to the input actions evoked by those subexpressions). As a consequence, whenever the evaluator is about to execute an input action, it first has to decide whether there will occur in the future another input action that should be done first according to the specified 'syntactic' partial order. That problem, however, is undecidable.

Our considerations finally led to the following requirement: the left to right order in the datatype definition is the order in which the input actions for a value of that type occur. So, for type `List Text`, the input of `Cons "a" (Cons "b" (Cons "c" Nil))` will always occur in the following order:

`Cons; "a"; Cons; "b"; Cons; "c"; Nil.`

If the evaluator is about to execute an input action for `head (tail ?inp)`, or more precisely for $?\texttt{inp}_{Cons_1,Cons_0}$, but the value of `head ?inp` (more precisely $?\texttt{inp}_{Cons_0}$) has not yet been read in, then first an additional input action is forced for this latter value.

In view of our simulation in pure Haskell, this requirement (that the order is derived from the datatype definition) is no obstacle for an implementer. In view of the examples we have tried, this requirement is good enough for quite a few programs, interactive games like hangman and tictactoe in particular.

The proposal for Output

8 Order of output. As motivated in the introduction we distinguish between the final *result* of a functional program, which normally is shown on the standard output channel, and other additional *output* for which the computation is not primarily intended. We stick to this nomenclature of *result* and *output* in the sequel. For example, the output may show the intermediate states of the computation whereas the result may consist of only the final state, or some value derived from that. Since the output is not the primary purpose of the computation, it can —almost by definition— only be intended to help the user in deciding on the next input action. Therefore we make the assumption that each output is related to and should come before the next input action. This assumption solves the problem of getting control over the order in which the output actions occur.

However, there still is the problem of *how* to associate output to input actions. This cannot be done within the declaration of the input constants, like the prompt messages, because at that place the state of the computation is not in scope. Output of a state has to be specified at a place where all variables making up the state are in scope. The problem, then, is that at such a place there is no control over which subexpression will cause which input action: for `rev ?inp`, for example, the evaluation of `rev (tail xs)` and the input action for $?\texttt{inp}_{Cons_1}$ in particular, will cause a forced input action for $?\texttt{inp}_{Cons_0}$.

9 Output construct. In this paper we use the following notation for an output directive:

`OUT` *expr*0`:` *expr*1 `->` *expr*

Disregarding input/output the expression behaves the same as *expr*; the reduction rule reads:

(`OUT` *expr*0 : *expr*1 `->` *expr*) $\rightarrow$ *expr*

As a side-effect, however, *expr*0 is evaluated until an input constant $?x$ is obtained or another closed normal form; in the former case the evaluator is required to store the pair ($?x$, *expr*1) and remember it for later use. Moreover, all output directives together have the following additional effect: when somewhere sometime an input action is about to be done for some input constant $?x$, then first the value of *expr*1 of each (or: just one) stored pair ($?x$, *expr*1) is shown as output. If no such pair has been stored so far, no output is shown.

> (So, exactly which of the output directives for $?x$ produce output, very much depends on the order of evaluation and is therefore highly unpredictable or indeterminate. This is, however, not unreasonable from the viewpoint that output is only intended to help the user to make up his mind for the subsequent input: if several outputs have been expressed in the program for the same input constant, they should all be equally helpful. The only reason for having several such output directives for the same $?x$ is that the subexpression whose evaluation causes the input action for $?x$ might be quite hard to locate and may differ between various program executions.
>
> An alternative stipulation would be that the least or most recently evaluated output directive is chosen, instead of all evaluated output directives. All these alternatives are equally easy to implement.)

Note that there is some unpredictability or nondeterminacy in the input/output effect of a program, but this doesn't affect the *result*. The expressions keep to be referentially transparent (meaning that the order of evaluation doesn't matter for the *result* and equational reasoning is applicable). This even applies to the *expr*0 and *expr*1 part of output directives! So, there is even no objection to the presence of input/output within the *expr*0 and *expr*1 parts, as far as the result of the program is considered.

Simulation of the proposed i/o mechanism in pure Haskell

We thoroughly discuss the simulation of the input mechanism, and then conclude with a brief discussion of the output directives and the special i/o windows.

10 Example program. We will explain the simulation of the proposed input mechanism in pure Haskell in terms of a "typical" example. This should make clear how to generalize the simulation to other datatypes. The example is the one we have discussed before:

```
data List a = Nil | Cons a (List a)
type Xs     = List Text -- dedicated to the input list and its tails
rev xs      = ifNil xs "." (rev (tl xs) ++ hd xs ++ hd xs)
?inp       :: Xs  where prompts as suggested at the end of paragraph 6
main        = rev ?inp
```

In order to show that the translation is quite systematic, we have used intermediate functions `hd`, `tl` and `ifNil` (which can be eliminated easily in exchange for one application of a `case` construct). For ease of presentation we do all destructions of a list into its constituents by a dedicated function `caseXs`:

```
caseXs xs nil cons = case xs of Nil -> nil; Cons h t -> cons h t
--
hd      :: Xs -> Text
tl      :: Xs -> Xs
ifNil   :: Xs -> a -> a -> a
++      :: Text -> Text -> Text
rev     :: Xs -> Text
--
hd xs    = caseXs xs (error "hd: nil") (\h t -> h)
tl xs    = caseXs xs (error "tl: nil") (\h t -> t)
ifNil xs e1 e2 = caseXs xs e1 (\h t -> e2)
```

11 The overall translation. The main idea for the translation is to define an "Extended IO" monad `IOE`, with which we can implicitly store and update *not only* the standard input and output streams of the program, *but also* several partially read-in values. Furthermore, we will define later some "primitives" upon which the translation is based, namely:

```
type IOE a  =  the Extended IO monad
type Xs'    =  replacement for Xs, dedicated to input lists
type Text'  =  replacement for Text, dedicated to input texts
inp'        :: IOE Xs'  --  represents the input constant ?inp
run         :: IOE a -> IO a
caseXs'     :: IOE Xs' -> IOE a -> (Text' -> Xs' -> IOE a) -> IOE a
caseText'   :: Text' -> IOE Text
```

The program will then be translated as follows (with an explanation afterwards):

```
hd'        :: IOE Xs' -> IOE Text
tl'        :: IOE Xs' -> IOE Xs'
ifNil'     :: IOE Xs' -> IOE a -> IOE a -> IOE a
+++        :: IOE Text -> IOE Text -> IOE Text
rev'       :: IOE Xs' -> IOE Text
--
hd' xs     = caseXs' xs (error "hd: nil") (\h t -> caseText' h)
tl' xs     = caseXs' xs (error "tl: nil") (\h t -> return t)
ifNil' xs e1 e2 = caseXs' xs e1 (\h t -> e2)
xs +++ ys  = do xs <- xs; ys <- ys; return (xs ++ ys)
--
```

```
rev' xs    = ifNil' xs (return ".") (rev'(tl' xs) +++ hd' xs +++ hd' xs)
main'      = rev' inp'
--
main       = run main'
```

The translation from the program with our input mechanism to the pure Haskell program, full of primes, proceeds as follows. The very first step is to remove the declaration of the input constant `?inp` and to replace all its occurrences by `inp'`; only the main call `rev ?inp` has such an occurrence. This replacement gives a type conflict, since `?inp` has type `Xs` whereas `inp'` has type `IOE Xs'`. Each following step eliminates the type conflicts introduced in the previous step, by "lifting" expressions in the *immediate* context of the type conflict with `IOE`, and changing `Xs` into `Xs'` if necessary. So, in the second step, `rev :: Xs -> Text` in the (just changed) main call `rev inp'` is replaced by a new function `rev' :: IOE Xs' -> IOE Text` (a "lifted" version of `rev`), so that the new main call `rev' inp'` has no type conflicts any more. Function `rev'` initially gets the same body as `rev`, so new type conflicts arise in this function body. In order to eliminate these conflicts, we observe that inside the –initial– body of `rev'` precisely `ifNil`, `tl`, and `hd` get an `IOE Xs'` argument rather than one of type `Xs`, so we need to replace these functions by lifted versions `ifNil'`, `tl'`, and `hd'`. Having done so, we get new type conflicts, namely for the applications of `++` in the body of the new reversal function, and in the bodies of the newly defined functions `ifNil'`, `tl'`, and `hd'` (whose bodies are initially copied from the original functions). This process goes on and on, until the entire program has been lifted from t to `IOE` t (with some necessary replacements of `Xs` by `Xs'`), whereupon it is subject to `run`. Apart from changing types, the following steps may be taken in the process:

- Replacing and lifting `caseXs` to `caseXs'`.
- Replacing and lifting a subexpression e `::` t to `return` e `::` `IOE` t.
- Bringing down a value x `::` `IOE` t to t by `do`-ing `x <-` x (as in `+++`)
  (not useful for $t =$ `Xs'` since the only primitive for dealing with `Xs'`-values is `caseXs'`).

Function `caseXs'` needs an argument of type `(Text'->Xs'->IOE a)`, corresponding to parameter `cons` of type `(Text->Xs->a)` in `caseXs`. This forces us to consider values of type `Text'`:

- Type `Text'` can be converted to `IOE Text` by `caseText'`.

Recall that the order of the actual input actions will be fixed and derived from the datatype definition. Therefore, the order in which the statements `xs<-xs` and `ys<-ys` occur in the body of `+++` is irrelevant: it has no effect on the result produced by the program!

The reader may wonder where the actual input actions are expressed. The answer is: in `caseXs'` and `caseText'`. These are the only functions that inspect an input constant or its constituents, so just these functions see whether their argument is not yet known and need to be read in. These two functions are defined in paragraph 13; in the coming paragraph we define `IOE`, `Xs'`, `Text'`, `inp'`, and `run`.

12 Storage of input values. To define the "Extended IO" monad `IOE` we start with recalling the well-known Store Transformation `ST`:

```
data ST s a  = InST (s -> IO (a, s))  -- s stands for: store
```

It is obvious how to establish that `ST s` is an instance of `Monad`; we omit the details. For our particular example program, we only need to store a single partially read-in list, so we take:

```
type Store   = ListP TextP
type IOE a   = ST Store a
```

In the general case, if there are n distinct input constants in the program, `Store` is an n-tuple of partially read-in values. The "partially known lists" `ListP a` and "partially known texts" `TextP` are very much like the original ones, the main difference being an additional alternative, signalling 'Unknown':

```
data ListP a = ListU | NilK | ConsK a (ListP a)
data TextP   = TextU | TextK Text
```

Thus K, P, U stand for completely Known, Partially known, and completely Unknown. The input constant `?inp`, and each of its constituents, is now represented by a *path* (a list of zeros and ones) into the `Store` where each `1` stands for 'take the tail' and a final `0` stands for 'take the head':

```
type Xs'    = [Int] -- all of the form 1:1:...:1:[]
type Text'  = [Int] -- all of the form 1:1:...:1:0:[]
inp'        = []    -- represents entire stored list
```

The interpretation of such a path is an easy lookup of the value in the store, and we also must be able to update the store with a new value at a given path:

```
lookupList  ::                Xs'   -> Store -> ListP TextP
lookupText  ::                Text' -> Store -> TextP
updList     :: ListP TextP -> Xs'   -> Store -> Store
updText     :: TextP       -> Text' -> Store -> Store
```

Their definitions are easily completed:

```
lookupList  []       list        = list
lookupList  (1:path) (ConsK h t) = lookupList path t
⋮
updList new []       ListU       = new
updList new (1:path) (ConsK h t) = updList new path t
⋮
```

Finally, we run an `IOE` program by giving it the completely unknown list `ListU` as store:

```
run (InIOE f) = do (a, store') <- f ListU; return a
```

13 Issuing the actual input commands. Recall the design decision that `caseText'` and `caseXs'` are the only functions that inspect an input constant, and access their constituents. They do so by looking up that value in the store, and may then find that the value is `TextU` or `ListU`: not yet known. In that case, an input command has to be issued, and the store has to be updated with the value read in. So, for `caseText'` we have the following definition:

```
caseText' :: Text' -> IOE Text
caseText' path = InIOE (\store -> do
   case lookupText path store of
      TextK s ->  return (s, store)
      TextU   ->  do s <- input"Elt?"; return (s, updText s path store))
```

The `IO` function `input "Elt?"` may behave identical to `getLine`, thus reading from `stdin`, or it may simulate that behavior in a separate window as explained in paragraph 15.

We turn now to `caseXs'`. We shorten its text a bit by using the inverse of `InIOE`:

```
outIOE (InIOE f) = f
```

For an easy understanding (especially regarding the roles of parameters `nil` and `cons`) we recall the definition of `caseXs`:

```
caseXs xs nil cons = case xs of Nil -> nil; Cons h t -> cons h t
```

Here is the definition of `caseXs'`, an explanation follows below:

```
caseXs' :: IOE Xs' -> IOE a -> (Text' -> Xs' -> IOE a) -> IOE a
caseXs' (InIOE f) nil cons = InIOE (\store -> do
  let cons' =  almost identical to cons (see below)
  (path, store') <- f store
  case lookupList path store' of
     NilK      -> outIOE nil store'
     ConsK h t -> outIOE (cons' (path++[0]) (path++[1])) store'
     ListU     -> do line <- input "Nil/[Cons]?"
                     if line /= "" && head line == 'N'
                        then outIOE nil (updList NilK path store')
                        else outIOE (cons' (path++[0]) (path++[1]))
                               (updList (ConsK TextU ListU) path store'))
```

Assuming for the moment that `cons'` equals `cons`, we see that, just as in `caseText'`, a lookup is done in the store and, in case the result is known (`NilK` or `ConsK`), parameters `nil` and `cons` are invoked just as in `caseXs`. In case the lookup results in `ListU`, then, apparently, access is requested to a part of the input constant that has not yet been read in. So, an input command is issued, and depending on the input, either `nil` or `cons` is as yet invoked.

With the assumption `cons'` = `cons`, the order in which the constituents are read in, is the order in which the computation needs the values. In order to force a specific fixed order, in particular the left to right order derived from the datatype definition, we define `cons'` in such a way that it first inspects the left neighbor constituent, reading in that value if applicable (in the same way as `caseText'`), and further behaves as the original `cons`. So, the line "`let cons' =`" reads as follows:

```
let cons' path0 path1 = InIOE (\store -> do
     case lookupText path0 store of
        TextK s -> outIOE (cons path0 path1) store
        TextU   -> do s <- input "Elt?"
                      outIOE (cons path0 path1) (updText s path0 store))
```

This completes the simulation in pure Haskell of our proposed input mechanism.

14 The output directives. To simulate the proposed output mechanism in pure Haskell, we extend the store `Store` of the previous paragraphs in such a way that for each and every constituent of the input constant another value (called "the output value") may be stored besides the value-to-be-read-in. Such an output value is to be given as output just before the input constant constituent to which it belongs, is read in. So, apart from straightforward adaptations of all functions that deal with the store, functions `caseText'` and `caseXs'` in particular need a major modification: when issuing an input command, they should first inspect whether an output value is present for the constituent, and if so, present that output value to the user. The evaluation of an output directive just stores the output value at the appropriate place in the store; thanks to lazy evaluation, the output value will not yet be reduced to normal form — this will only happen when it is actually presented to the user by `caseText'` and `caseXs'`.

15 Separate windows for input and output. We sketch a definition of a function `input` that works like the standard function `getLine`, except that it generates a new, temporary, window from which it reads its value. Assuming a Unix-like environment, and using `system` from module `System`, the definition reads as follows:

```
input prompt = do
  _      <- system (Unix command to create a file in.tmp as a pipe)
  _      <- system ("xterm   -T "++prompt++"   -e ./script_in   &")
  line   <- fGetLine "in.tmp"
  ...
  return line
```

The second `system` statement generates a new window, having `prompt` in the title bar and in which script `script_in` is executed. The script takes care that everything typed in into the window, is put into file `in.tmp` (simply by `cat > in.tmp`). Auxiliary function `fGetLine` should open file `in.tmp`, read a line, and close it; the line is returned as the final function result. Just before returning the final result, the window has to be destroyed. This can be done by `system` by issuing the Unix command `kill` with a suitable process identifier; the script `script_in` can easily write this process id to a special file (`echo $$ > pid.tmp`), so that at the end the process id can be got from the file (and file `pid.tmp` can be removed). There are various beautifications possible, which we shall not discuss here. The idea for output windows is similar.

References

[1] Peter Achten. *Interactive Functional Programs – Models, Methods, and Implementation.* PhD thesis, University of Nijmegen, Nijmegen, The Netherlands, 1996.

[2] Peter Achten and Rinus Plasmeijer. Interactive functional objects in Clean. In Chris Clack, Kevin Hammond, and Tony Davie, editors, *Implementation of Functional Languages*, volume 1467 of *Lect. Notes in Comp. Sc.*, pages 304–321, Berlin, 1998. Springer.

[3] R.S. Bird. *Introduction to Functional Programming using Haskell, second edition.* Prentice Hall Europe, London, 1998.

[4] Simon Peyton Jones (ed.) et al. Report on the Programming Language Haskell 98, A Non-strict, Purely Functional Language. Technical report, Haskell community, February 1999. http://haskell.org.

[5] Simon Thompson. *Haskell: The Craft of Functional Programming.* Addison-Wesley, Harlow, England, 1996.