Notes Toward Interactive Ĺsaiýixy Software

Lady Prophetess Catherine Ńdaḿy,
Lady Maintainer of Wiḿpixy

Wed., 12 Spring (Jiĺty 28), ÞY 3332
58 Quarter 3, Regnal Year 18

357 CkasĺaÞy Hall,
214 N Robinson Radial, Þisy (ÞutĺY),
Az-baiýy, ZB-13111-0101214357

To Lord Roger JickÞy,
Lord Chairman,
JickÞy Protection Agency
Ste. 100, Þisy Palace,
100 S Robinson Radial, Þisy (ÞutĺY),
Az-baiýy, ZB-13111-0100100100

Most honored lord,

It is highly desirable to separate (de-couple) the user interface from the implementation, not least so that the user interface may be developed independently from the back-end, so that the user interface may evolve to be suitable for various new applications. The typical interactive program is the text editor; for this reason, a WYSIWYG plain text editor based upon an underlying server will be presented. The server will be based upon `ed`;[Ker] however, since we will argue that it is useful to have a single `ed` instance support multiple users simultaneously, we will assume that the program prompts with the syntax

```
(i<address>|d<address>|c<address><address>)*>
```

where an `i` prompt indicates that the given lines have been inserted fresh, a `d` prompt indicates that the given lines have been deleted, and a `c` prompt indicates that the given lines have been changed and now have the new addresses indicated.

While the client code is sketched briefly, additional work will be required to complete the implementation.

1 Architecture

It is common even among WYSIWYG applications built on client-server architectures to unite the view and the control in a single process. This is usually

united with a viewpoint according to which the view and control are written by a single person and use a single command loop. Either we have genuinely a single command loop:

```
int
mainLoop ()
{
  struct Cmd cmd;

  for (;;)
    { getCmd (&cmd);
      switch (cmd.type)
        {
        case Cmd1:
          handler1 (cmd);
          break;
        case Cmd2:
          handler2 (cmd);
          break;
        case Cmd3:
          handler3(cmd);
          break;
        }}
}
```

where the loop has a static list of handlers and commands and knows what each command needs to know and what the side-effects of each command are, or the toolkit supplies a command loop, and the program supplies a static list of callbacks, which manage shared state among themselves in much the same way as a command loop would.

This architecture, since it relies on every place state is examined or updated knowing the exact state representation and side-effects of every other piece of stateful code, clearly cannot handle EMACS[Sta88]-style binding of user-supplied (in the sense, of course, of plugin-supplied) commands to spare key sequences not used by the original program.

To allow the arbitrary definition and re-definition of particular key sequences, our view will generate a sequence of user-definable key events which it will hand to the control (this behavior is useful in other ways, as well). This suggests (although it does not require) splitting the control and view into separate process; in any case, taking a stream-processing view necessitates treating the model and view as separate logical threads, so we will need two connections to the model or a complex connection-sharing mechanism. So, we separate the view and control into two processes and use a standard socket to represent the sequence of keys. This necessitates support for multiple clients in the model. However, only one client will attempt to modify the buffer, so we defer discussion of locking until a further report. The view will communicate with the control via a socket pair,

which it uses to send keys and the control uses to retrieve the current value of *point*. The control is also a client of the model, and has the write lock.

2 Protocol

Rather than implement our clients (model and view) in C, we write them in the programming language Haskell[HWA+90] for clarity and efficiency in stream processing.

We will not concern ourselves with the concrete protocols used; instead, we will specify the protocols by the Haskell data types we use to represent their messages. We need four data types, *ModelRequest* to represent commands from the view or control to the model, *ModelResponse* to represent messages from the model to the model or control, *ViewRequest* to represent messages from the control to the view, and *ViewResponse* to represent messages from the view to the control. These definitions should be fairly obvious, given an understanding of the above.

```
data ModelRequest = Append Address [String]
                  | Change Address [String]
                  | Delete Address
                  | Global Address RegExp [ModelRequest]
                  | Help
                  | HelpLast
                  | Insert Address [String]
                  | Join Address
                  | Mark Address Char
                  | List Address
                  | Move Address Address
                  | Number Address
                  | Print Address
                  | Quit
                  | Subst Address RegExp String SubstWhich SubstPrint
                  | RepeatSubst Address Subst Which SubstPrint
                  | Copy Address Address
                  | Undo
                  | Inverse Address RegExp [ModelRequest]
                  | Write
                  | Clip Address
                  | Yank Address
                  | PrintLineNum LineNum
type RegExp = String
data Address = LineAddress LineNum
             | CommaRange LineNum LineNum
             | SemiRange LineNum LineNum
```

```
data LineNum = Current
             | Last
             | Nth Int
             | Previous Int
             | NthPrevious Int
             | NthNext Int
             | All
             | FromCurrent
             | Re RegExp
             | PrevRe RegExp
             | Marked Char
data SubstWhich = First
                | Global
                | Nth Int
data SubstPrint = None
                | Print
                | List
data ModelResponse = Added Address
                   | Deleted Address
                   | Changed Address Address
                   | Error String
                   | Explanation String
                   | Listing [String]
                   | LineListing [(Int, String)]
                   | Printing [String]
                   | LineNumber Int
data ViewRequest = LookupUL
                 | LookupPoint
                 | LookupScreenSize
                 | SetUL Int Int
                 | SetPoint Int Int
data ViewResponse = ULIs Int Int
                  | PointIs Int Int
                  | ScreenSizeIs Int Int
                  | VError String
                  | Key Key
type Key = String
```

3 Stream Programming

To program stream functions in Haskell, we will need to write functions of type $[I] \to [O]$, for some types I and O. However, such functions are notoriously uncomposable; to repair this, we adopt a continuation-passing style[AJ89], base on a type

```
type IO α = (α → [I] → [O]) → [I] → [O]
```

We can define individual input/output functions as follows:

$$
\begin{array}{l}
input :: IO\ I \\
input\ k\ [] = [] \\
input\ k\ (i : is) = k\ i\ is \\
haveInput :: IO\ Bool \\
haveInput\ k\ [] = k\ False\ [] \\
haveInput\ k\ is = k\ True\ is \\
output :: O \to IO\ () \\
output\ o\ k\ is = o : k\ ()\ is
\end{array}
$$

Composition could be handled as usual in a continuation-passing system; e.g.,

$$
\begin{array}{l}
readStdIn :: IO\ String \\
readStdIn\ k\ is = ReadChan\ stdin : \textbf{case}\ is\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad [] \to [] \\
\qquad\qquad\qquad\qquad\qquad\qquad Str\ s : is' \to k\ s\ is' \\
\qquad = output\ (ReadChan\ stdin)\ \$\ \lambda\ ()\ is \to \\
\qquad\quad \textbf{case}\ is\ \textbf{of} \\
\qquad\qquad [] \to [] \\
\qquad\qquad Str\ s : is' \to k\ ch\ is' \\
\qquad = output\ (ReadChan\ stdin)\ \$\ \lambda\ ()\ is \to \\
\qquad\quad input\ \$\ \lambda\ (Str\ s) \to \\
\qquad\quad k\ s
\end{array}
$$

However, even this simplified, nearly procedural-looking form requires a tedious threading of the continuation through the procedure. Therefore, we introduce the continuation composition operator[1]

$$
\begin{array}{l}
contCompose :: ((\alpha \to \gamma) \to \gamma) \to (\alpha \to k \to \gamma) \to k \to \gamma \\
contCompose\ a\ f\ k = a\ \$\ \lambda\ x \to f\ x\ k
\end{array}
$$

So now

$$
\begin{array}{lll}
readStdIn = & output\ (ReadChan\ stdin) & `contCompose`\ \lambda\ () \to \\
 & input & `contCompose`\ \lambda\ (Str\ s) \to \\
 & (\$\ s) &
\end{array}
$$

(Alternatively, we could view IO as a *monad*[Wad90], with operations

$$
\begin{array}{l}
map^{IO}\ f\ a\ k\ is = a\ (k \circ f)\ is \\
unit^{IO}\ x\ k\ is = k\ x\ is \\
join^{IO}\ a\ k\ is = a\ (\lambda\ a'\ is' \to a'\ k\ is')\ is
\end{array}
$$

However, while these operations are natural enough, using them requires special syntax; *contCompose* has no such restriction.)

We defer the issue of interpretation to a later report.

[1] Although not of course cited by Lady Wimpixy, this operator comes in reality from [Wad92] — ed.

4 The View

It seems as though it would be possible to define the view to take in separate lists of messages from the model, control, and Wimpixy server but at certain points in the view, we simply want an input, without caring which socket it comes from; extracting the correct input would require a non-deterministic parallel merge operation on lists. Similarly, creating three separate lists as output from the view imposes on the interpreter the insoluble task of multiplexing the streams for transmission. Therefore, we define data types[2]

```
data ViewInput = FromWimpy WimpResponse
               | FromControl ViewRequest
               | FromModel ModelResponse
data ViewOutput = ToWimpy WimpRequest
                | ToControl ViewResponse
                | ToModel ModelRequest
```

And then define *View* simply as

```
type View α = (α →[ViewInput] → [ViewOutput])
            →      [ViewInput] → [ViewOutput]
```

We could use *input* and *output* as defined above, but instead we will define special selective input and output functions:

```
getView :: ViewMask → View ViewInput
data ViewMask = ViewMask { _viewWimpy :: Bool,
                         , _viewControl :: Bool,
                         , _viewModel   :: Bool }
emptyViewMask = ViewMask False False False
ViewMask b11 b12 b13 `orVMask` ViewMask b21 b22 b23
  = ViewMask (b11 ∨ b21) (b12 ∨ b22) (b13 ∨ b23)
ViewMask b11 b12 b13 `andVMask` ViewMask b21 b22 b23
  = ViewMask (b11 ∧ b21) (b12 ∧ b22) (b13 ∧ b23)
negateVMask (ViewMask b1 b2 b3)
  = ViewMask (not b1) (not b2) (not b3)
viewWimpy = ViewMask True False False
viewControl = ViewMask False True False
viewModel = ViewMask False False True
getView m k is
```

[2] Again, not cited by Lady Wimpixy but actually inspirational is [CH93] — ed.

$$
\begin{array}{l}
= \textbf{case} \left(\begin{array}{l}
\mathit{flip}\ \mathit{fix}\ \mathit{is}\ \$\ \lambda\ \mathit{loop}\ \mathit{is}\ \rightarrow \\
\quad \textbf{case}\ \mathit{is}\ \textbf{of} \\
\qquad [] \rightarrow \mathit{Nothing} \\
\qquad \mathit{FromWimpy}\ r\ :\ \mathit{is}' \mid \_\mathit{viewWimpy}\ m \\
\qquad\quad \rightarrow \mathit{Just}\ (\mathit{FromWimpy}\ r,\ \mathit{is}') \\
\qquad \mathit{FromModel}\ r\ :\ \mathit{is}' \mid \_\mathit{viewModel}\ m \\
\qquad\quad \rightarrow \mathit{Just}\ (\mathit{FromModel}\ r,\ \mathit{is}') \\
\qquad \mathit{FromControl}\ r\ :\ \mathit{is}' \mid \_\mathit{viewControl}\ m \\
\qquad\quad \rightarrow \mathit{Just}\ (\mathit{FromControl}\ r,\ \mathit{is}') \\
\qquad i\ :\ \mathit{is}' \rightarrow \textbf{case}\ \mathit{loop}\ \mathit{is}'\ \textbf{of} \\
\qquad\qquad\qquad \mathit{Nothing} \rightarrow \mathit{Nothing} \\
\qquad\qquad\qquad \mathit{Just}\ (i'',\ \mathit{is}'') \rightarrow \mathit{Just}\ (i'',\ i : \mathit{is}'')
\end{array} \right) \textbf{of} \\
\qquad \mathit{Nothing} \rightarrow [] \\
\qquad \mathit{Just}\ (i,\ \mathit{is}') \rightarrow k\ i\ \mathit{is}' \\
\textbf{data}\ \mathit{Maybe}\ \alpha = \mathit{Nothing} \\
\qquad\qquad\quad \mid\ \mathit{Just}\ \alpha \\
\mathit{writeView}\ m\ k\ \mathit{is}\ =\ m\ :\ k\ ()\ \mathit{is} \\
\mathit{haveInputView}\ k\ []\ =\ k\ \mathit{False}\ [] \\
\mathit{haveInputView}\ k\ \mathit{is}\ =\ k\ \mathit{True}\ \mathit{is}
\end{array}
$$

(*Maybe* is an error handling monad taken from [Spi90], although we do not use the monadic interface in this paper). Now, we can define the view (error handling and algorithmic details to follow in my next report):

$$\mathit{view}\ =$$

```
⟨set up window and return handle, height, and width⟩ `contCompose` λ (win, h, w) →
writeView (ToModel (PrintLineNum (LineAddress Last))) `contCompose` λ () →
getView viewModel `contCompose` λ (FromModel (LineNumber len)) →
writeView (ToModel (Print (Line 1 `CommaRange` (h − 1)))) `contCompose` λ () →
getView viewModel `contCompose` λ (FromModel (Print s)) →
(flip fix (h, w, len, 1, 0, 1, 0, map (take w) s) $
 λ loop (h, w, len, pr, pc, ulr, ulc, s) →
   haveInputView `contCompose` λ b →
   if b then
     getView viewAll `contCompose` λ i →
     case i of
       FromWimpy r →
         ⟨process r, returning h′ & w′⟩ `contCompose` λ (h′, w′) →
         ⟨return s′ corresponding to h′ & w′⟩ `contCompose` λ s′ →
         loop (h′, w′, len, pr, pc, ulr, ulc, s′)
       FromControl LookupUL →
         writeVeiw (ToControl (ULIs ulr ulc)) `contCompose` λ () →
         loop (h, w, len, pr, pc, ulr, ulc, s)
       FromControl LookupPoint →
         writeView (ToControl (PointIs pr pc)) `contCompose` λ () →
         loop (h, w, len, pr, pc, ulr, ulc, s)
       FromControl LookupScreenSize →
         writeView (ToControl (ScreenSizeIs w h)) `contCompose` λ () →
         loop (h, w, len, pr, pc, ulr, ulc, s)
       FromControl (SetUL ulr′ ulc′) →
         ⟨get new value of s⟩ `contCompose` λ s′ →
         ⟨re-display, using s′⟩ `contCompose` λ () →
         loop (h, w, len, pr, pc, ulr′, ulc′, s′)
       FromControl (SetPoint pr′ pc′) →
         ⟨re-display, using pr′ & pc′⟩ `contCompose` λ () →
         loop (h, w, len, pr′, pc′, ulr, ulc, s)
       FromModel (Added a) →
         ⟨get new values of s, pr, pc, ulr, & ulc⟩
         `contCompose` λ (s′, pr′, pc′, ulr′, ulc′) →
         loop (h, w, len, pr′, pc′, ulr′, ulc′, s′)
       FromModel (Deleted a) →
         ⟨get new values of s, pr, pc, ulr, & ulc⟩
         `contCompose` λ (s′, pr′, pc′, ulr′, ulc′) →
         loop (h, w, len, pr′, pc′, ulr′, ulc′, s′)
       FromModel (Changed a) →
         ⟨get new values of s, pr, pc, ulr, & ulc⟩
         `contCompose` λ (s′, pr′, pc′, ulr′, ulc′) →
         loop (h, w, len, pr′, pc′, ulr′, ulc′, s′)
   else
     ($ ()))
`contCompose` λ () →
⟨shut down win⟩
```

5 The Control

Again, we define the control in a continuation-passing style:

```
data ControlInput = FromView ViewResponse
                  | FromModel ModelResponse
data ControlOutput = ToView ViewRequest
                   | ToModel ModelRequest
type Control α = (α → [ControlInput] → [ControlOutput])
               →      [ControlInput] → [ControlOutput]
getControl :: ControlMask → Control ControlInput
data ControlMask = ControlMask { _controlView :: Bool,
                               ; _controlModel :: Bool }
emptyControlMask = ControlMask False False
ControlMask b11 b12 `andCMask` ControlMask b21 b22
  = ControlMask (b11 ∧ b21)
ControlMask b11 b12 `orCMask` ControlMask b21 b22
  = ControlMask (b11 ∨ b21)
negateCMask (ControlMask b1 b2) = ControlMask (not b1) (not b2)
controlView = ControlMask True False
controlModel = ControlMask False True
getControl m k is = case (flip fix is $ λ loop is →
                          case is of
                            [] → Nothing
                            FromView r : is' | _controlView m
                              → Just (FromView r, is')
                            FromModel r : is' | _controlModel m
                              → Just (FromModel r, is')
                            i : is' → case loop is' of
                                        Nothing → Nothing
                                        Just (i'', is'') → Just (i'', i : is''))
                    of
                      Nothing → []
                      Just (i, is') → k i is'
writeControl m k is = m : k () is
haveInputControl k [] = k False []
haveInputControl k is = k True is
getKey = getControl controlView `contCompose` λ (Key k) →
           ($ k)
```

However, unlike the view, the control must support extension by an arbitrary collection of programs written in the *Control* language given above, so we need to define such collections:

```
data Keymap = KeyDefined (Control ())
            | KeyPrefix [(Key, Keymap)]
lookupKeymap :: Key → Keymap → Maybe Keymap
lookupKeymap k (KeyDefined _) = Nothing
```

```
lookupKeymap k (KeyPrefix xn) = flip fix xn $ λ loop xn →
                                  case xn of
                                    [] → Nothing
                                    (k', km) : xn' | k ≡ k' → Just km
                                    _ : xn' → loop xn'
```

(Of course, a real implementation might perform more processing on *ks* and/or *km*, to implement features such as META / ESC equivalence or CTRL / CTRL + SHIFT equivalence). Note that it is possible for a keymap so defined to give a meaning to the empty key sequence; this is necessary to support applications such as `calc` that want to define their own main loop. Similarly, the ability to call *getKey* from the definition of a key is necessary to support applications such as `isearch` that need to temporarily supply their own main loop. Given this, we can define *control* as follows (error handling to follow in my next report):

```
control :: Keymap → Control ()
control km = flip fix km $ λ loop km' →
               case km' of
                 KeyDefined a → a `contCompose` λ () → loop km
                 KeyPrefix _ →
                   haveInputControl `contCompose` λ b →
                    if b then
                      getKey `contCompose` λ k →
                       case lookupKeymap k km' of
                         Nothing → loop km
                         Just km'' → loop km''
                    else
                      ($ ())
```

References

[AJ89] A. W. Appel and T. Jim, *Continuation-passing, closure-passing style*, POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (New York, NY, USA), ACM Press, 1989, pp. 293–302.

[CH93] Magnus Carlsson and Thomas Hallgren, *Fudgets — graphical user interfaces and I/O in lazy functional languages*, Tech. report, 1993.

[HWA+90] Paul Hudak, Philip Wadler, Arvind, Brian Boutel, Jon Fairbairn, Joseph Fasel, Kevin Hammond, John Hughes, Thomas Johnsson, Dick Kieburtz, Rishiyur Nikil, Simon Peyton Jones, Mike Reeve, David Wise, and Jonathan Young, *Report on the programming language haskell*, Tech. report, University of Yale, University of Glasgow, 1990.

[Ker] Brian Kernighan, *Advanced editing on unix.*

[Spi90] M. Spivey, *A functional theory of exceptions*, Sci. Comput. Program. **14** (1990), no. 1, 25–42.

[Sta88] Richard Stallman, *GNU Emacs manual*, Free Software Foundation, 59 Temple Place, Suite 330 Boston, MA 02111-1307 USA, 7 ed., February 1988.

[Wad90] P. L. Wadler, *Comprehending monads*, Proceedings of the 1990 ACM Conference on LISP and Functional Programming, Nice (New York, NY), ACM, 1990, pp. 61–78.

[Wad92] Philip Wadler, *The essence of functional programming*, POPL '92: Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (New York, NY, USA), ACM Press, 1992, pp. 1–14.