Monads for Interactive Ĺsaiýixy Software

Lady Prophetess Catherine Ńdaḿy,
Lady Maintainer of Wiḿpixy

Fri., 42 Spring (29 Taglÿ), ÞY 3332
88 Q3, 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,

Contrary to previous reports, monads[Wad90] are tremendously useful in the development of interactive software. Therefore, to explain in what way this is true, a review of monads is in order, starting with the notion of a *functor*.

A Haskell functor is a type function F with the signature

$$\begin{array}{l}
\textbf{type}\ F\ \alpha \\
map^F\ ::\ (\alpha \rightarrow \beta) \rightarrow F\ \alpha \rightarrow F\ \beta \\
map^F\ id\ =\ id \\
map^F\ (f \circ g)\ =\ map^F\ f \circ map^F\ g
\end{array}$$

A Haskell monad is a type function M with the signature

$$\begin{array}{l}
\textbf{type}\ M\ \alpha \\
map^M\ ::\ (\alpha \rightarrow \beta) \rightarrow M\ \alpha \rightarrow M\ \beta \\
unit^M\ ::\ \alpha \rightarrow M\ \alpha \\
join^M\ ::\ M\ (M\ \alpha) \rightarrow M\ \alpha \\
map^M\ id\ =\ id \\
map^M\ (f \circ g)\ =\ map^M\ f \circ map^M\ g \\
unit^M \circ f\ =\ map^M\ f \circ unit^M \\
join^M \circ map^M\ (map^M\ f)\ =\ map^M\ f \circ join^M \\
join^M \circ unit^M\ =\ id \\
join^M \circ map^M\ unit^M\ =\ id
\end{array}$$

$$join^M \circ map^M\, join^M = join^M \circ join^M$$

As noted by previous reports, programming with this signature requires special syntax. However, the *contCompose* operator from our previous report is actually a general monad operator, introduced in the general case by [Wad92], which can of course be used in lieu of special syntax (as it was in our previous paper). The monad generalization of *contCompose* is $bind^M$, defined for a general monad M by

$$\begin{aligned}
&bind^M :: M\,\alpha \to (\alpha \to M\,\beta) \to M\,\beta\\
&bind^M\, a\, f = join^M\,(map^M\, f\, a)
\end{aligned}$$

From which we can prove three very useful laws:

$$unit^M\, x \text{ `}bind^M\text{` } f = f\, x\; a \text{ `}bind^M\text{` } unit^M = a\,(a \text{ `}bind^M\text{` } h) \text{ `}bind^M\text{` } k = a \text{ `}bind^M\text{` } \lambda\, x \to h$$

Proof. We will prove the first law, and omit the (similar) proofs of the others to save space.

$$\begin{aligned}
&\;\; unit^M\, x \text{ `}bind^M\text{` } f\\
&= join^M\,(map^M\, f\,(unit^M\, x))\\
&= join^M\,(unit^M\,(f\, x))\\
&= f\, x \quad \square
\end{aligned}$$

Even more interestingly, we can define map^M and $join^M$ in terms of $bind^M$:

$$map^M\, f\, a = a \text{ `}bind^M\text{` } unit^M \circ f\; join^M\, a = a \text{ `}bind^M\text{` } id$$

Furthermore, we can prove the 8 laws above from these three and the two definitions above:

Proof. We prove $map^M\,(f \circ g) = map^M\, f \circ map^M\, g$, and omit the other proofs to save space.

$$\begin{aligned}
&\;\; map^M\,(f \circ g)\, a\\
&= a \text{ `}bind^M\text{` } unit^M \circ f \circ g\\
&= a \text{ `}bind^M\text{` } \lambda\, x \to unit^M\,(f\,(g\, x))\\
&= a \text{ `}bind^M\text{` } \lambda\, x \to unit^M\,(g\, x) \text{ `}bind^M\text{` } unit^M \circ f\\
&= (a \text{ `}bind^M\text{` } unit^M \circ g) \text{ `}bind^M\text{` } unit^M \circ f\\
&= map^M\, f\,(map^M\, g\, a) \quad \square
\end{aligned}$$

Furthermore, we can prove the natural transformation laws without parametricity using the $bind^M$ laws; the same proof will work in any language, regardless of what non-parametric operations it has (of which Haskell has in fact two: $\perp$ and *strict*, which if not standard is certainly common).

1 The View

We incorporate the protocol, *ViewInput*, and *ViewOutput* data types from our previous report, except that we re-define the following constructors from *ModelResponse* (we need to interpret them *without round trips*):

$$
\begin{array}{lll}
\textbf{data}\ \mathit{ModelResponse} = & \mathit{Added\ Int\ Int} & \text{— New line numbers} \\
 \mid & \mathit{Deleted\ Int\ Int} & \text{— Old line numbers} \\
 \mid & \mathit{Changed\ Int\ Int\ Int} & \text{— Old line numbers, new length} \\
 \vdots & &
\end{array}
$$

However, viewing *View* as a monad allows us to incorporate the view state into the monad, as such:

$$
\begin{array}{l}
\textbf{type}\ \mathit{View}\ \alpha = [\mathit{ViewInput}] \to \mathit{ViewState} \\
\qquad \to (\alpha \to [\mathit{ViewInput}] \to \mathit{ViewState} \to [\mathit{ViewOutput}]) \\
\qquad \to [\mathit{ViewOutput}] \\
\mathit{unit}^{\mathit{View}}\ x\ \mathit{is}\ s\ k\ =\ k\ x\ \mathit{is}\ s \\
(a\ \text{`}\mathit{bind}^{\mathit{View}}\text{`}\ f)\ \mathit{is}\ s\ k\ =\ a\ \mathit{is}\ s\ \$\ \lambda\ x\ \mathit{is}'\ s'\ \to\ f\ x\ \mathit{is}'\ s'\ k \\
\textbf{data}\ \mathit{ViewState}\ =\ \mathit{ViewState} \left\{ \begin{array}{ll}
\mathit{viewHeight} & :: \mathit{Int}, \\
\mathit{viewWidth} & :: \mathit{Int}, \\
\mathit{viewLength} & :: \mathit{Int}, \\
\mathit{viewPointRow} & :: \mathit{Int}, \\
\mathit{viewPointColumn} & :: \mathit{Int}, \\
\mathit{viewULRow} & :: \mathit{Int}, \\
\mathit{viewULColumn} & :: \mathit{Int}, \\
\mathit{viewText} & :: \mathit{Maybe}\ [\mathit{String}]
\end{array} \right\} \\
\mathit{getVState}\ ::\ (\mathit{ViewState}\ \to\ \alpha)\ \to\ \mathit{View}\ \alpha \\
\mathit{getVState}\ f\ \mathit{is}\ s\ k\ =\ k\ (f\ s)\ \mathit{is}\ s \\
\mathit{modifyVState}\ ::\ (\mathit{ViewState}\ \to\ \mathit{ViewState})\ \to\ \mathit{View}\ () \\
\mathit{modifyVState}\ f\ \mathit{is}\ s\ k\ =\ k\ ()\ \mathit{is}\ (f\ s)
\end{array}
$$

Furthermore, we adopt a better technique for filtering inputs than in the last report; in particular, we now process prompts from the model in *getView*.

$$
\begin{array}{l}
\mathit{process}\ n_1\ n_2\ \mathit{len}\ =\ \mathit{modifyVState}\ \$\ \lambda\ s\ \to \\
\qquad s \left\{ \begin{array}{ll}
\mathit{viewLength} & = \mathit{viewLength}\ s\ -\ (n_2\ +\ 1\ -\ n_1)\ +\ \mathit{len}, \\
\mathit{viewPointRow} & = \mathit{adjust}\ (\mathit{viewPointRow}\ s), \\
\mathit{viewULRow} & = \mathit{adjust}\ (\mathit{viewULRow}\ s), \\
\mathit{viewText} & = \mathit{Nothing}
\end{array} \right\}
\end{array}
$$

```
 where
   adjust i = i
              + if i ≥ n2 then
                    len − (n2 + 1 − n1)
                else
                    0
              + if n2 > i ∧ i ≥ n1 then
                    − (max 0 (len − (i − n)1))
                else
                    0
getView :: (ViewInput → Bool) → View ViewInput
getView p =
  (λ is s k → case is of
                  [] → []
                  i : is′ → k i is′ s) ‘bind^View‘ λ i →
  case i of
    FromModel (Added n1 n2) →
      process (n1 − 1) (n1 − 1) (n2 + 1 − n1) ‘bind^View‘ λ () →
      getView p
    FromModel (Deleted n1 n2) →
      process (n1 n2 0) ‘bind^View‘ λ () →
      getView p
    FromModel (Changed n1 n2 len) →
      process n1 n2 len ‘bind^View‘ λ () →
      getView p
    _ | p i → unit^View i
    _ → getView p ‘bind^View‘ λ i′ →
          (λ is s k → k () (i′ : is) s) ‘bind^View‘ λ () →
          unit^View i
haveInputView :: View Bool
haveInputView k is s =
  k (null (filter (λ m → case m of
                             FromModel (Added _ _) → False
                             FromModel (Deleted _ _) → False
                             FromModel (Changed _ _ _) → False
                             _ → True) is)) is s
isFromModel, isFromControl, isFromWimpy :: ViewInput → Bool
isFromModel (FromModel _) = True
isFromModel _ = False
isFromControl (FromControl _) = True
isFromControl _ = False
isFromWimpy (FromWimpy _) = True
isFromWimpy _ = False
(∨*), (∧*) :: (α → Bool) → (α → Bool) → α → Bool
(f ∨* g) x = f x ∨ g x
(f ∧* g) x = f x ∧ g x
putView :: ViewOutput → View ()
```

```
putView m is s k = m : k () is s
getVText :: View String
getVText =
  getVState viewText ‘bind^View‘ λ mb →
  case mb of
    Just s → unit^View s
    Nothing →
      fix $ λ loop →
        getVState viewULRow ‘bind^View‘ λ ulr →
        getVState viewULColumn ‘bind^View‘ λ ulc →
        getVState viewHeight ‘bind^View‘ λ h →
        getVState viewWidth ‘bind^View‘ λ w →
        modifyVState (λ st → st{viewText = Just ""}) ‘bind^View‘ λ () →
        putView (ToModel (Print (ulr, ulr + h − 1))) ‘bind^View‘ λ () →
        getView isFromModel ‘bind^View‘ λ (Printing s) →
        getVState viewText ‘bind^View‘ λ mb →
        case mb of
          Just "" →
            let
              s' = map (take w . drop ulc) s
            in
            setVState (λ st → st{viewText = Just s'}) ‘bind^View‘ λ () →
            unit^View s'
          Nothing → loop
```

Having defined this, the view itself becomes easier to define:

$$
\begin{array}{l}
\mathit{view}\ =\\
\quad \langle\text{set up window, storing height and width and returning handle}\rangle\ `\mathit{bind}^{\mathit{View}}`\ \lambda\,\mathit{win}\ \rightarrow\\
\quad \mathit{putView}\ (\mathit{ToModel}\ (\mathit{PrintLineNum}\ (\mathit{LineAddress}\ \mathit{Last})))\ `\mathit{bind}^{\mathit{View}}`\ \lambda\,()\ \rightarrow\\
\quad \mathit{getView}\ \mathit{isFromModel}\ `\mathit{bind}^{\mathit{View}}`\ \lambda\,(\mathit{FromModel}\ (\mathit{LineNumber}\ \mathit{len}))\ \rightarrow\\
\quad \mathit{modifyVState}\ (\lambda\,s\ \rightarrow\ s\{\mathit{viewLength}\ =\ \mathit{len}\})\ `\mathit{bind}^{\mathit{View}}`\ \lambda\,()\ \rightarrow\\
\quad \left(\begin{array}{l}
\mathit{fix}\ \$\ \lambda\,\mathit{loop}\ \rightarrow\\
\quad \mathit{haveInputView}\ `\mathit{bind}^{\mathit{View}}`\ \lambda\,b\ \rightarrow\\
\quad \mathbf{if}\ b\ \mathbf{then}\\
\quad\quad \mathit{getView}\ (\mathit{const}\ \mathit{True})\ `\mathit{bind}^{\mathit{View}}`\ \lambda\,i\ \rightarrow\\
\quad\quad \mathbf{case}\ i\ \mathbf{of}\\
\quad\quad\quad \mathit{FromWimpy}\ r\ \rightarrow\\
\quad\quad\quad\quad \langle\text{process } r\rangle\ `\mathit{bind}^{\mathit{View}}`\ \lambda\,()\ \rightarrow\\
\quad\quad\quad\quad \mathit{loop}\\
\quad\quad\quad \mathit{FromControl}\ \mathit{LookupUL}\ \rightarrow\\
\quad\quad\quad\quad \mathit{getVState}\ \mathit{viewULRow}\ `\mathit{bind}^{\mathit{View}}`\ \lambda\,\mathit{ulr}\ \rightarrow\\
\quad\quad\quad\quad \mathit{getVState}\ \mathit{viewULColumn}\ `\mathit{bind}^{\mathit{View}}`\ \lambda\,\mathit{ulc}\ \rightarrow\\
\quad\quad\quad\quad \mathit{putView}\ (\mathit{ToControl}\ (\mathit{ULIs}\ \mathit{ulr}\ \mathit{ulc}))\ `\mathit{bind}^{\mathit{View}}`\ \lambda\,()\ \rightarrow\\
\quad\quad\quad\quad \mathit{loop}\\
\quad\quad\quad \mathit{FromControl}\ \mathit{LookupPoint}\ \rightarrow\\
\quad\quad\quad\quad \mathit{getVState}\ \mathit{viewPointRow}\ `\mathit{bind}^{\mathit{View}}`\ \lambda\,\mathit{pr}\ \rightarrow\\
\quad\quad\quad\quad \mathit{getVState}\ \mathit{viewPointColumn}\ `\mathit{bind}^{\mathit{View}}`\ \lambda\,\mathit{pc}\ \rightarrow\\
\quad\quad\quad\quad \mathit{putView}\ (\mathit{ToControl}\ (\mathit{PointIs}\ \mathit{pr}\ \mathit{pc}))\ `\mathit{bind}^{\mathit{View}}`\ \lambda\,()\ \rightarrow\\
\quad\quad\quad\quad \mathit{loop}\\
\quad\quad\quad \mathit{FromControl}\ \mathit{LookupScreenSize}\ \rightarrow\\
\quad\quad\quad\quad \mathit{getVState}\ \mathit{viewHeight}\ `\mathit{bind}^{\mathit{View}}`\ \lambda\,h\ \rightarrow\\
\quad\quad\quad\quad \mathit{getVState}\ \mathit{viewWidth}\ `\mathit{bind}^{\mathit{View}}`\ \lambda\,w\ \rightarrow\\
\quad\quad\quad\quad \mathit{putView}\ (\mathit{ToControl}\ (\mathit{ScreenSizeIs}\ w\ h))\ `\mathit{bind}^{\mathit{View}}`\ \lambda\,()\ \rightarrow\\
\quad\quad\quad\quad \mathit{loop}\\
\quad\quad\quad \mathit{FromControl}\ (\mathit{SetUL}\ \mathit{ulr}'\ \mathit{ulc}')\ \rightarrow\\
\quad\quad\quad\quad \mathit{modifyVState}\ \left(\lambda\,s\ \rightarrow\ s\left\{\begin{array}{l}\mathit{viewULRow}\ =\ \mathit{ulr}',\\ \mathit{viewULColumn}\ =\ \mathit{ulc}',\\ \mathit{viewText}\ =\ \mathit{Nothing}\end{array}\right\}\right)\ `\mathit{bind}^{\mathit{View}}`\ \lambda\,()\ \rightarrow\\
\quad\quad\quad\quad \mathit{loop}\\
\quad\quad\quad \mathit{FromControl}\ (\mathit{SetPoint}\ \mathit{pr}'\ \mathit{pc}')\ \rightarrow\\
\quad\quad\quad\quad \mathit{modifyVState}\ \left(\lambda\,s\ \rightarrow\ s\left\{\begin{array}{l}\mathit{viewPointRow}\ =\ \mathit{pr}',\\ \mathit{viewPointColumn}\ =\ \mathit{pc}'\end{array}\right\}\right)\ `\mathit{bind}^{\mathit{View}}`\ \lambda\,()\ \rightarrow\\
\quad\quad\quad\quad \langle\text{re-display, using } \mathit{pr}'\ \&\ \mathit{pc}'\rangle\ `\mathit{bind}^{\mathit{View}}`\ \lambda\,()\ \rightarrow\\
\quad\quad\quad\quad \mathit{loop}\\
\quad \mathbf{else}\\
\quad\quad \mathit{unit}^{\mathit{View}}\ ()
\end{array}\right)\\
\quad `\mathit{bind}^{\mathit{View}}`\ \lambda\,()\ \rightarrow\\
\quad \langle\text{shut down } \mathit{win}\rangle
\end{array}
$$

Translating the control into monadic form requires only the systematic substitutions $\mathit{contCompose}\ \rightarrow\ \mathit{bind}^{\mathit{Control}}$ and $(\$\ x)\ \rightarrow \mathit{unit}^{\mathit{Control}}\ x$; the translation is omitted.

References

[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.