Update monads

Gabe Dijkstra

July 25th, 2018

London Haskell Meetup

Update monads

- Introduced by Danel Ahman and Tarmo Uustalu in 2014 ("Update monads: cointerpreting directed containers.")
- Generalises reader, writer and state monad
- Separate state into updates and their interpretation

Reader versus writer versus state

- Reader monad
  - computations that depend on reading from somewhere
  - example: reading configuration
  - $Reader\ r\ a = r \to a$
- Writer monad
  - computations that write to somewhere
  - example: writing to logs
  - $Writer\ m\ a = (a, m)$
- State monad
  - computations that read from somewhere but also write to it
  - example: mutable variables
  - $State\ s\ a = s \to (a, s)$

Update monads combine reader and writer monads via monoid actions

Monoids and monoid actions

Monoid action interprets monoid elements as transformations:

$$\underline{\text{class}}\ \mathit{Action}\ p\ s\ \underline{\text{where}}$$
$$\mathit{act} :: p \to s \to s$$

satisfying:

- $\mathit{act}\ \mathit{mempty} = \mathit{id}$
- $\mathit{act}\ (x \diamond y) = \mathit{act}\ y \circ \mathit{act}\ x$

Examples:

- $\mathit{applyStyle} :: \mathit{Style} \to \mathit{Diagram} \to \mathit{Diagram}$ (from `diagrams` package)
- $\mathit{interp} :: [\mathit{Instr}] \to \mathit{Env} \to \mathit{Env}$ (interpreting instructions)

Update monad – definition

We define:

$$\underline{\text{newtype}}\ \mathit{Update}\ p\ s\ a = \mathit{Update}\ \{\, \mathit{runUpdate} :: s \to (p, a) \,\}$$

Note that:

$$\begin{aligned} \mathit{Update}\ p\ s\ a &\cong \mathit{Reader}\ s\ (\mathit{Writer}\ p\ a) \\ &\cong \mathit{Reader}\ s\ (p, a) \\ &\cong s \to (p, a) \end{aligned}$$

Update monad – functor instance

$$\underline{\text{newtype}}\ \mathit{Update}\ p\ s\ a = \mathit{Update}\ \{\ \mathit{runUpdate} :: s \rightarrow (p, a)\ \}$$

$$\begin{array}{l}\underline{\text{instance}}\ \mathit{Functor}\ (\mathit{Update}\ p\ s)\ \underline{\text{where}} \\ \quad \mathit{fmap}\ f\ (\mathit{Update}\ g) = \mathit{Update}\ \$\ \lambda s \rightarrow \underline{\text{let}}\ (p, a) = g\ s \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \underline{\text{in}}\ (p, f\ a)\end{array}$$

Update monad – applicative instance

```haskell
newtype Update p s a = Update { runUpdate :: s → (p, a) }
```

```haskell
instance (Monoid p, Action p s) ⇒ Applicative (Update p s) where
  pure a = Update $ λs → (mempty, a)
  (Update f) <*> (Update g) = Update $ λs →
    let (p, h)  = f s
        (p′, a) = g (act p s)
    in  (p <> p′, h a)
```

Update monad – monad instance

$$\underline{\text{newtype}}\ \mathit{Update}\ p\ s\ a = \mathit{Update}\ \{\mathit{runUpdate} :: s \to (p, a)\}$$

$$
\begin{array}{l}
\underline{\text{instance}}\ (\mathit{Monoid}\ p, \mathit{Action}\ p\ s) \Rightarrow \mathit{Monad}\ (\mathit{Update}\ p\ s)\ \underline{\text{where}} \\
\quad (\mathit{Update}\ f) \gg\!= g = \mathit{Update}\ \$\ \lambda s \to \\
\qquad \underline{\text{let}}\ (p, a) \ \ = f\ \ s \\
\qquad \qquad h' \qquad = g\ a \\
\qquad \qquad (p', b) = \mathit{runUpdate}\ h'\ (\mathit{act}\ p\ s) \\
\qquad \underline{\text{in}}\ \ (p \diamond p', b)
\end{array}
$$

Reader as update monad

Choose r as the state and $()$ as the monoid:

$$\underline{\text{type}}\ \mathit{UpdateReader}\ r\ a = \mathit{Update}\ ()\ r\ a$$

$$\begin{aligned}&\mathit{updateAsk} :: \mathit{UpdateReader}\ r\ r\\&\mathit{updateAsk} = \mathit{Update}\ \$\ \lambda r \to ((), r)\end{aligned}$$

Example program:

$$\begin{aligned}&\mathit{testReader} :: \mathit{String}\\&\mathit{testReader} = \mathit{snd}\ \$\ \mathit{runUpdate}\ \mathit{prog}\ \texttt{"hello"}\\&\quad \underline{\text{where}}\ \mathit{prog} = \mathit{updateAsk}\end{aligned}$$

Output:

$$\begin{aligned}&\mathit{testReader}\\&\gg \texttt{"hello"}\end{aligned}$$

Writer as update monad

Choose () as the state and m as the monoid:

$$\underline{\text{type}}\ \mathit{UpdateWriter}\ m\ a = \mathit{Update}\ m\ ()\ a$$

with trivial action:

$$\begin{array}{l}\underline{\text{instance}}\ \mathit{Monoid}\ m \Rightarrow \mathit{Action}\ m\ ()\ \underline{\text{where}}\\ \quad \mathit{act}\ \_\ \_ = ()\\ \mathit{updateTell} :: m \rightarrow \mathit{UpdateWriter}\ m\ ()\\ \mathit{updateTell}\ m = \mathit{Update}\ \$\ \lambda\_ \rightarrow (m, ())\end{array}$$

Writer as update monad – example

Example program:

$$\begin{array}{l}
\mathit{testWriter} :: [\mathit{Int}] \\
\mathit{testWriter} = \mathit{fst}\ \$\ \mathit{runUpdate}\ \mathit{prog}\ () \\
\quad \underline{\text{where}} \\
\qquad \mathit{prog} = \underline{\text{do}} \\
\qquad\quad \mathit{updateTell}\ [1] \\
\qquad\quad \mathit{updateTell}\ [2,3] \\
\qquad\quad \mathit{updateTell}\ [4]
\end{array}$$

Output:

$$\begin{array}{l}
\mathit{testWriter} \\
\gg [1,2,3,4]
\end{array}$$

State as update monad

Choose s as the state and $Last\ s$ as the monoid:

$$\underline{\text{newtype}}\ Last\ s = Last\ \{ getLast :: Maybe\ a \}$$

$$\begin{aligned}
&\underline{\text{instance}}\ Monoid\ (Last\ s)\ \underline{\text{where}} \\
&\quad mempty = Last\ Nothing \\
&\quad x \diamond (Last\ Nothing) = x \\
&\quad _ \diamond y = y
\end{aligned}$$

$$\begin{aligned}
&\underline{\text{instance}}\ Action\ (Last\ s)\ s\ \underline{\text{where}} \\
&\quad act\ (Last\ Nothing)\ s' = s' \\
&\quad act\ (Last\ (Just\ s))\ _ = s
\end{aligned}$$

$$\underline{\text{type}}\ UpdateState\ s\ a = Update\ (Last\ s)\ s\ a$$

(Note: $UpdateState\ s\ a$ is not isomorphic to $State\ s\ a$.)

State as update monad – *get* and *set*

Choose s as the state and $\mathit{Last}\ s$ as the monoid:

$$\underline{\text{type}}\ \mathit{UpdateState}\ s\ a = \mathit{Update}\ (\mathit{Last}\ s)\ s\ a$$

get and *set*:

$$\begin{aligned}
&\mathit{updateGet} :: \mathit{UpdateState}\ s\ s \\
&\mathit{updateGet} = \mathit{Update}\ \$\ \lambda s \to (\mathit{mempty}, s) \\
&\mathit{updateSet} :: s \to \mathit{UpdateState}\ s\ () \\
&\mathit{updateSet}\ s = \mathit{Update}\ \$\ \lambda_ \to (\mathit{Last}\ (\mathit{Just}\ s), ())
\end{aligned}$$

State as update monad – example program

Example program:

$$
\begin{array}{l}
\mathit{testState} :: (\mathit{Last\ Int}, \mathit{Int}) \\
\mathit{testState} = \mathit{runUpdate\ prog}\ 1 \\
\quad \underline{\text{where}}\ \mathit{prog} = \underline{\text{do}} \\
\qquad x \leftarrow \mathit{updateGet} \\
\qquad \mathit{updateSet}\ 2 \\
\qquad \mathit{updateSet}\ 3 \\
\qquad \mathit{pure}\ x
\end{array}
$$

Output:

$$
\begin{array}{l}
\mathit{testState} \\
\gg (\mathit{Last}\ \{\mathit{getLast} = \mathit{Just}\ 3\}, 1)
\end{array}
$$

State-logging monad

Choose s as the state and $[s]$ as the monoid:

$$\underline{\text{type}}\ \mathit{StateLogging}\ s\ a = \mathit{Update}\ [s]\ s\ a$$

with action:

$$\begin{array}{l} \underline{\text{instance}}\ \mathit{Action}\ [s]\ s\ \underline{\text{where}} \\ \quad \mathit{act}\ \mathit{ss}\ s = \mathit{last}\ (s : \mathit{ss}) \end{array}$$

State-logging monad – *get* and *set*

Choose s as the state and $[s]$ as the monoid:

$$\underline{\text{type}}\ \mathit{StateLogging}\ s\ a = \mathit{Update}\ [s]\ s\ a$$

get and set:

$$\begin{aligned}
&\mathit{logGet} :: \mathit{LogState}\ s\ s \\
&\mathit{logGet} = \mathit{Update}\ \$\ \lambda s \to (\mathit{mempty}, s) \\
&\mathit{logSet} :: s \to \mathit{LogState}\ s\ () \\
&\mathit{logSet}\ s = \mathit{Update}\ \$\ \lambda_ \to ([s], ())
\end{aligned}$$

State-logging monad – example

Example program:

$$
\begin{aligned}
&\mathit{testLogState} :: ([\mathit{Int}], \mathit{Int})\\
&\mathit{testLogState} = \mathit{runUpdate}\ \mathit{prog}\ 1\\
&\quad \underline{\text{where}}\\
&\qquad \mathit{prog} = \underline{\text{do}}\\
&\qquad\quad x \leftarrow \mathit{logGet}\\
&\qquad\quad \mathit{logSet}\ 2\\
&\qquad\quad \mathit{logSet}\ 3\\
&\qquad\quad \mathit{pure}\ x
\end{aligned}
$$

Output:

$$
\begin{aligned}
&\mathit{testLogState}\\
&\gg ([2, 3], 1)
\end{aligned}
$$

Stack monad

Choose the state to be a stack and the monoid to be stack operations:

$$\underline{\text{data}}\ \mathit{StackOp}\ s = \mathit{Push}\ s \mid \mathit{Pop}$$

$$\begin{aligned}
&\mathit{applyOp} :: [s] \to \mathit{StackOp}\ s \to [s] \\
&\mathit{applyOp}\ \mathit{tack}\qquad\ (\mathit{Push}\ s) = s : \mathit{tack} \\
&\mathit{applyOp}\ (s : \mathit{tack})\ \mathit{Pop}\qquad = \mathit{tack} \\
&\mathit{applyOp}\ []\qquad\quad\ \mathit{Pop}\qquad = \mathit{error}\ \texttt{"Whoops!"}
\end{aligned}$$

$$\begin{aligned}
&\underline{\text{instance}}\ \mathit{Action}\ [\mathit{StackOp}\ s]\ [s]\ \underline{\text{where}} \\
&\quad \mathit{act}\ \mathit{stackOps}\ \mathit{stack} = \mathit{foldl}\ \mathit{applyOp}\ \mathit{stack}\ \mathit{stackOps}
\end{aligned}$$

$$\underline{\text{type}}\ \mathit{StackMonad}\ s\ a = \mathit{Update}\ [\mathit{StackOp}\ s]\ [s]\ a$$

Stack monad – push and pop

Choose the state to be a stack and the monoid to be stack operations:

$$\underline{\text{data}}\ \mathit{StackOp}\ s = \mathit{Push}\ s \mid \mathit{Pop}$$
$$\underline{\text{type}}\ \mathit{StackMonad}\ s\ a = \mathit{Update}\ [\mathit{StackOp}\ s]\ [s]\ a$$

Push and pop:

$$\mathit{stackPush} :: s \to \mathit{StackMonad}\ s\ ()$$
$$\mathit{stackPush}\ s = \mathit{Update}\ \$\ \lambda\_ \to ([\mathit{Push}\ s], ())$$
$$\mathit{stackPop} :: \mathit{StackMonad}\ s\ s$$
$$\mathit{stackPop} = \mathit{Update}\ \$\ \lambda s \to ([\mathit{Pop}], \mathit{head}\ s)$$

Stack monad – example program

Example:

$$
\begin{aligned}
&\mathit{testStack} :: ([\mathit{StackOp}\ \mathit{Int}], \mathit{Int})\\
&\mathit{testStack} = \mathit{runUpdate}\ \mathit{prog}\ [\,]\\
&\quad \underline{\text{where}}\ \mathit{prog} = \underline{\text{do}}\\
&\qquad \mathit{stackPush}\ 1\\
&\qquad \mathit{stackPush}\ 2\\
&\qquad x \leftarrow \mathit{stackPop}\\
&\qquad \mathit{stackPush}\ (x * x)\\
&\qquad \mathit{stackPop}
\end{aligned}
$$

Output:

$$
\begin{aligned}
&\mathit{testStack}\\
&\gg ([\mathit{Push}\ 1, \mathit{Push}\ 2, \mathit{Pop}, \mathit{Push}\ 4, \mathit{Pop}], 4)
\end{aligned}
$$

Update monad examples

Monad	State	Monoid	Monoid action
Reader	r	$()$	$\lambda\_\ s \rightarrow s$
Writer	$()$	m	$\lambda\_\ \_ \rightarrow ()$
"State"	s	$Last\ s$	$\lambda(Last\ x)\ s = fromMaybe\ s\ x$
State-logging	s	$[s]$	$\lambda ss\ s \rightarrow last\ (s : ss)$
Stack	[s]	$[StackOp\ s]$	$\lambda ops\ s \rightarrow foldl\ applyOp\ s\ ops$

Story so far

Update monads are like state monads, but separate state into:

- a monoid of updates
- a monoid action which interprets these updates

How do we combine *Reader* and *Writer* to get *State*?

Combining functors to get a monad

$$\begin{aligned} \mathit{State}\ s\ a &\cong \mathit{Reader}\ s\ (\mathit{Writer}\ s\ a) \\ &\cong \mathit{Reader}\ s\ (a, s) \\ &\cong s \to (a, s) \end{aligned}$$

Where does the monad structure on *State s* come from?

Combining functors to get a monad

There is a special relationship between the *Reader s* and *Writer s* functors:

- $to :: (Writer\ s\ a \to b) \to (a \to Reader\ s\ a)$
- $from :: (a \to Reader\ s\ a) \to (Writer\ s\ a \to b)$
- $to \circ from = id$
- $from \circ to = id$

Combining functors to get a monad

There is a special relationship between the *Reader s* and *Writer s* functors:

- $to :: ((a, s) \to b) \to (a \to s \to a)$
- $from :: (a \to s \to b) \to ((a, s) \to b)$
- $to \circ from = id$
- $from \circ to = id$

Combining functors to get a monad

There is a special relationship between the *Reader s* and *Writer s* functors:

- $curry :: ((a, s) \to b) \to (a \to s \to a)$
- $uncurry :: (a \to s \to b) \to ((a, s) \to b)$
- $curry \circ uncurry = id$
- $uncurry \circ curry = id$

Combining functors to get a monad – adjunctions

Two functors $f, g :: * \to *$ have an adjunction if we have:

- $to :: (f\ a \to b) \to (a \to g\ b)$
- $from :: (a \to g\ b) \to (f\ a \to b)$
- $to \circ from = id$
- $from \circ to = id$

If we have an adjunction, we can write instance $Monad\ (Compose\ g\ f)$

Recall: newtype $Compose\ g\ f = Compose\ (g\ (f\ a))$

In our case on $State\ s\ \cong\ Compose\ (Reader\ s)\ (Writer\ s)$

How do we combine *Reader* and *Writer* to get *Update*?

Combining monads to get a monad

Can we write the following instance?

$$\underline{\text{instance}}\ (Monad\ m, Monad\ n) \Rightarrow Monad\ (Compose\ m\ n)\ \underline{\text{where}}$$

Combining monads to get a monad – *pure*

Goal: define $pure_{m \circ n} :: a \to m\ (n\ a)$

Combining monads to get a monad – *pure*

Goal: define $pure_{m \circ n} :: a \to m\ (n\ a)$

$$pure_{m \circ n} = pure_m \circ pure_n$$

Combining monads to get a monad – *join*

Goal: define $join_{m \circ n} :: m\ (n\ (m\ (n\ a))) \to m\ (n\ a)$

Have:

- $join_m : m\ (m\ a) \to m\ a$
- $join_n : n\ (n\ a) \to n\ a$

Combining monads to get a monad – *join*

Goal: define $join_{m \circ n} :: m\ (n\ (m\ (n\ a))) \rightarrow m\ (n\ a)$

Have:

- $join_m : m\ (m\ a) \rightarrow m\ a$
- $join_n : n\ (n\ a) \rightarrow n\ a$

We are stuck... If only we could swap m and n...

Combining monads to get a monad – distributive laws

$\underline{\text{class}}$ $\mathit{DistributiveLaw}\ m\ n$ $\underline{\text{where}}$
$\quad \mathit{distribute} :: n\ (m\ a) \rightarrow m\ (n\ a)$

Combining monads to get a monad – distributive laws

$\underline{\text{class}}$ $\mathit{DistributiveLaw}\ m\ n$ $\underline{\text{where}}$
  $\mathit{distribute} :: n\ (m\ a) \to m\ (n\ a)$

(… such that *distribute* respects monad laws of m and n …)

Combining monads to get a monad – distributive laws

$\underline{\text{class}}\ \mathit{DistributiveLaw}\ m\ n\ \underline{\text{where}}$
$\quad \mathit{distribute} :: n\ (m\ a) \rightarrow m\ (n\ a)$

(… such that *distribute* respects monad laws of m and n …)

- $\mathit{fmap}\ \mathit{distribute} :: m\ (n\ (m\ (n\ a))) \rightarrow m\ (m\ (n\ (n\ a)))$
- $\mathit{join}_m :: m\ (m\ (n\ (n\ a))) \rightarrow m\ (n\ (n\ a))$
- $\mathit{fmap}\ \mathit{join}_n :: m\ (n\ (n\ a)) \rightarrow m\ (n\ a)$

We can define $\mathit{join}_{m \circ n}$ as:

- $\mathit{fmap}\ \mathit{join}_n \circ \mathit{join}_m \circ \mathit{fmap}\ \mathit{distribute} :: m\ (n\ (m\ (n\ a))) \rightarrow m\ (n\ a)$

Combining monads to get a monad – *Distributive* vs *Traversable*

Compare:

$$\underline{\text{class}}\ (\mathit{Monad}\ m, \mathit{Monad}\ n) \Rightarrow \mathit{Distributive}\ m\ n\ \underline{\text{where}}$$
$$\quad \mathit{distribute} :: n\ (m\ a) \to m\ (n\ a)$$

with

$$\underline{\text{class}}\ (\mathit{Functor}\ t, \mathit{Foldable}\ t) \Rightarrow \mathit{Traversable}\ t\ \underline{\text{where}}$$
$$\quad \mathit{sequenceA} :: \mathit{Applicative}\ f \Rightarrow t\ (f\ a) \to f\ (t\ a)$$

Distributive laws for *Reader* and *Writer* monads

What do distributive laws for *Reader s* and *Writer p* monads look like?

$$
\begin{aligned}
& \mathit{Writer}\ p\ (\mathit{Reader}\ s\ a) \to \mathit{Reader}\ s\ (\mathit{Writer}\ p\ a) \\
\cong\ & (p, s \to a) \to s \to (p, a)
\end{aligned}
$$

Distributive laws for *Reader* and *Writer* monads

What do distributive laws for *Reader s* and *Writer p* monads look like?

$$\begin{aligned} & \mathit{Writer}\ p\ (\mathit{Reader}\ s\ a) \to \mathit{Reader}\ s\ (\mathit{Writer}\ p\ a) \\ \cong\ & (p, s \to a) \to s \to (p, a) \end{aligned}$$

Use monoid action:

$$\begin{aligned} & \mathit{distribute} :: (\mathit{Monoid}\ p, \mathit{Action}\ p\ s) \Rightarrow (p, s \to a) \to s \to (p, a) \\ & \mathit{distribute}\ (p, f)\ s = (p, f\ (\mathit{act}\ p\ s)) \end{aligned}$$

Distributive laws for *Reader* and *Writer* monads

Distributive laws for *Reader s* and *Writer p* monads

$\Leftrightarrow$

Monoid actions $p \to s \to s$

$\Leftrightarrow$

Update monads *Update p s*

Distributive laws

Why are distributive laws called distributive?

A ring consists of:

- a commutative group $(+, 0)$ (addition)
- a monoid $(\cdot, 1)$ (multiplication)
- distributivity:
  - $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$
  - $(a + b) \cdot c = (a \cdot c) + (b \cdot c)$

The free ring monad arises from a distributive law between the free commutative group monad and free monoid monad

Dependently-typed update monad

(in dependent pseudo-Haskell)
Given:

$$
\begin{aligned}
& S :: * \\
& P :: S \to * \quad \text{-- a type family over S} \\
& e :: (s :: S) \to P\ s \quad \text{-- units} \\
& act :: (s :: S) \to P\ s \to S \\
& (\diamond) :: \{s :: S\} \to (p :: P\ s) \to P\ (act\ s\ p) \to P\ s
\end{aligned}
$$

We define:

$$
\underline{\text{newtype}}\ Update\ a = Update\ \{ runUpdate :: (s :: S) \to (P\ s, a) \}
$$

Allows us to have different available updates depending on the state

Summary

- Update monads split state into:
  - a monoid of updates
  - a monoid action that interprets these updates
- Adjunctions tell us how to combine functors to get a monad
- Distributive laws tell us how to combine monads to get a monad
- State monad: via an adjunction between *Writer s* and *Reader s*
- Update monads: via distributive laws between *Reader s* and *Writer p*

Further reading:

- Danel Ahman and Tarmo Uustalu, "Update monads: cointerpreting directed containers.", TYPES 2014