Monad and side effects

CHUNG-CHIEH SHAN

Theme: To get what you want, say what you mean.
So, if you want to do something, say what doing it means.

1 WARM UP

Sub-theme: For modular reuse, abstract from similarities over differences.

2 SIDE EFFECTS

Basically, a side effect is something that a piece of code does besides turning input arguments into return values.

- State (OG)
  - $\mathit{state} \to (\mathit{value}, \mathit{state})$
  - Local vs global state
- Exception, Maybe
  - $\mathit{error} + \mathit{value}$
- Nondeterminism
  - set/multiset/list/pointed-set/distribution of values
- Input, Output
  - $\mathit{input} \to \mathit{value}$
  - $(\mathit{value}, \mathit{output})$
  - Interactivity
- …

3 MONADS

A type constructor M with two operations: [Wadler 1995]

$$
\begin{array}{lll}
\mathit{return} & :: \forall a. \quad a \to M\ a & \text{-- unit, eta} \\
(\gg\!=) & :: \forall a\ b.\ M\ a \to (a \to M\ b) \to M\ b & \text{-- bind, star}
\end{array}
$$

Monad laws:

$$
\begin{array}{ll}
\mathit{return}\ a \gg\!= k & = k\ a \\
m \gg\!= \mathit{return} & = m \\
m \gg\!= (\lambda a \to k\ a \gg\!= l) & = (m \gg\!= k) \gg\!= l
\end{array}
$$

Alternative definition with three operations:

$$
\begin{array}{lll}
\mathit{return} & :: \forall a. \quad a \to M\ a & \text{-- unit, eta} \\
\mathit{fmap} & :: \forall a\ b.\ (a \to b) \to M\ a \to M\ b & \text{-- functoriality} \\
\mathit{join} & :: \forall a. \quad M\ (M\ a) \to M\ a & \text{-- mu}
\end{array}
$$

4 TYPE CLASSES

Dictionaries of methods (even binary methods) [Wadler and Blott 1989]

- Passed implicitly
- Constructed automatically during type inference

4.1 Eq

Default method implementations

```
class Eq a where
  (==) :: a → a → Bool
  (/=) :: a → a → Bool

  x /= y  =  not  (x == y)
  x == y  =  not  (x /= y)

instance Eq Int    where    -- ...

instance Eq Char where    -- ...

instance (Eq a) ⇒ Eq [a] where
  []         == []         = True
  (x : xs) == (y : ys) = x == y && xs == ys
  _          == _          = False

instance (Eq a, Eq b) ⇒ Eq (a, b) where
  (a1, b1) == (a2, b2) = a1 == a2 && b1 == b2
```

Instance contexts (constraints)

Type-signature contexts (constraints)

```
elem :: (Eq a) ⇒ a → [a] → Bool
```

4.2 Ord

Class contexts (superclasses)

```
class (Eq a) ⇒ Ord a where
  (<), (⩽), (>), (⩾) :: a → a → Bool
  x < y = not (x == y) &&       (x ⩽ y)
  x > y = not (x == y) && not (x ⩽ y)
  x ⩾ y =       (x == y)  ||  not (x ⩽ y)
  -- ...
```

4.3 Show

```
class Show a where
  show :: a → String
  -- ...
```

4.4 Monad

Refined class hierarchy

$$
\begin{array}{l}
\textbf{class}\ \mathit{Functor}\ m\ \textbf{where}\\
\quad \mathit{fmap}\ :: (a \to b) \to m\ a \to m\ b\\
\textbf{class}\ (\mathit{Functor}\ m) \Rightarrow \mathit{Applicative}\ m\ \textbf{where}\\
\quad \mathit{pure}\ :: a \to m\ a\\
\quad (\langle * \rangle)\ :: m\ (a \to b) \to m\ a \to m\ b\\
\textbf{class}\ (\mathit{Applicative}\ m) \Rightarrow \mathit{Monad}\ m\ \textbf{where}\\
\quad \mathit{return} :: a \to m\ a\\
\quad (\gg\!=)\ :: m\ a \to (a \to m\ b) \to m\ b
\end{array}
$$

Easy superclass implementations

$$
\begin{array}{l}
\textbf{newtype}\ \mathit{State}\ s\ a = \mathit{State}\ \{\mathit{runState} :: s \to (a, s)\}\\
\textbf{instance}\ \mathit{Functor}\ (\mathit{State}\ s)\ \textbf{where}\ \mathit{fmap} = \mathit{liftM}\\
\textbf{instance}\ \mathit{Applicative}\ (\mathit{State}\ s)\ \textbf{where}\ \mathit{pure} = \mathit{return}; (\langle * \rangle) = \mathit{ap}\\
\textbf{instance}\ \mathit{Monad}\ (\mathit{State}\ s)\ \textbf{where}\\
\quad \mathit{return}\ a = \mathit{State}\ (\lambda s \to (a, s))\\
\quad m \gg\!= k\ = \mathit{State}\ (\lambda s \to \textbf{let}\ (a, s') = \mathit{runState}\ m\ s\\
\qquad\qquad\qquad\qquad\qquad\ \ \textbf{in}\ \mathit{runState}\ (k\ a)\ s')
\end{array}
$$

IO is an abstract *Monad* [Peyton Jones 2001]

5 DO NOTATION

Imperative intuition. For example, monad laws:

$$
\begin{array}{ll}
\textbf{do}\ \{x \leftarrow \mathit{return}\ a; k\ x\} & = k\ a\\
\textbf{do}\ \{x \leftarrow m; \mathit{return}\ x\} & = m\\
\textbf{do}\ \{a \leftarrow m; \textbf{do}\ \{b \leftarrow k\ a; l\ b\}\} & = \textbf{do}\ \{b \leftarrow \textbf{do}\ \{a \leftarrow m; k\ a\}; l\ b\}
\end{array}
$$

6 POLYMORPHISM ACROSS MONADS

Action combinators, useful for all monads

$$
\begin{array}{ll}
\mathit{traverse} & :: (\mathit{Monad}\ m) \Rightarrow (a \to m\ b) \to [a] \to m\ [b]\\
\mathit{traverse\_} & :: (\mathit{Monad}\ m) \Rightarrow (a \to m\ b) \to [a] \to m\ ()\\
\mathit{sequence} & :: (\mathit{Monad}\ m) \Rightarrow [m\ a] \to m\ [a]\\
\mathit{sequence\_} & :: (\mathit{Monad}\ m) \Rightarrow [m\ a] \to m\ ()
\end{array}
$$

$$
\begin{array}{ll}
\multicolumn{2}{l}{\textbf{import}\ \mathit{Control.Monad}}\\
\mathit{replicateM} & :: (\mathit{Monad}\ m) \Rightarrow \mathit{Int} \to m\ a \to m\ [a]\\
\mathit{replicateM\_} & :: (\mathit{Monad}\ m) \Rightarrow \mathit{Int} \to m\ a \to m\ ()\\
\mathit{filterM} & :: (\mathit{Monad}\ m) \Rightarrow (a \to m\ \mathit{Bool}) \to [a] \to m\ [a]\\
\mathit{foldM} & :: (\mathit{Monad}\ m) \Rightarrow (b \to a \to m\ b) \to b \to [a] \to m\ b
\end{array}
$$

Generalized to containers other than lists: small but powerful API for container implementation to provide

7 COMBINING SIDE EFFECTS

Monad transformers [Liang et al. 1995]

$$
\begin{array}{l}
\textbf{class}\ \mathit{MonadTrans}\ t\ \textbf{where}\\
\quad \mathit{lift} :: (\mathit{Monad}\ m) \Rightarrow m\ a \to t\ m\ a
\end{array}
$$

Monad transformer laws

$$\begin{aligned}
&lift\ (return\ a) = return\ a \\
&lift\ (m \gg\!= k)\ = lift\ m \gg\!= (lift \circ k)
\end{aligned}$$

Operations need to be lifted too

7.1 State and IO

Make state dependencies explicit and contained, in just another monad with a different dictionary

$$StateT\ s\ IO\ a = s \to IO\ (a, s)$$

7.2 State and exception

Different combinations have different meanings

$$\begin{aligned}
&StateT\ s\ Maybe\ a && = s \to Maybe\ (a, s) \\
&MaybeT\ (State\ s)\ a && = s \to (Maybe\ a, s) \\
&StateT\ s\ (MaybeT\ (State\ t))\ a && = s \to t \to (Maybe\ (a, s), t)
\end{aligned}$$

Stack of memory regions

7.3 State and nondeterminism

Different combinations have different meanings

$$\begin{aligned}
&StateT\ s\ [\,]\ a && = s \to [\,(a, s)\,] \\
&ListT\ (State\ s)\ a && = s \to ([\,a\,], s)
\end{aligned}$$

Lazy evaluation in state [Fischer et al. 2011]

7.4 Parsing

[Hutton and Meijer 1998]

$$StateT\ String\ [\,]\ a = String \to [\,(a, String)\,]$$

Everything follows from a few operations for *String* and nondeterminism:

$$\begin{aligned}
&\mathbf{instance}\ Monad\ Parser \\
&empty :: Parser\ a \\
&(\langle|\rangle)\ \ :: Parser\ a \to Parser\ a \to Parser\ a \\
&item\ \ \ :: Parser\ Char
\end{aligned}$$

Efficiency concerns are tricky to reason about, because the data type does not express them

$$\begin{aligned}
&empty \langle|\rangle\ p && = p \\
&p \langle|\rangle\ empty && = p \\
&p \langle|\rangle\ (q \langle|\rangle\ r) && = (p \langle|\rangle\ q) \langle|\rangle\ r \\
&empty \gg\!= f && = empty \\
&p \gg\!= \backslash_ \to empty && = empty \\
&(p \langle|\rangle\ q) \gg\!= f && = (p \gg\!= f) \langle|\rangle\ (q \gg\!= f) \\
&p \gg\!= (\lambda a \to f\ a \langle|\rangle\ g\ a) && = (p \gg\!= f) \langle|\rangle\ (p \gg\!= g)
\end{aligned}$$

7.5 Probability

[Ramsey and Pfeffer 2002]

$\mathit{WriterT}\ (\mathit{Product\ Double})\ [\,]\ a = [\,(a, \mathit{Product\ Double})\,]$

Everything follows from a few operations for *Product Double* and nondeterminism:

instance *Monad Dist*
$\mathit{empty} :: \mathit{Dist}\ a$
$(\langle|\rangle)\ :: \mathit{Dist}\ a \to \mathit{Dist}\ a \to \mathit{Dist}\ a$
$\mathit{factor} :: \mathit{Double} \to \mathit{Dist}\ ()$

8 NEURAL NETS

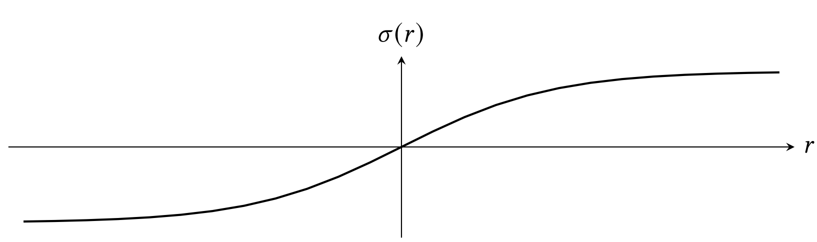
Examples	$\boldsymbol{x}_i \mapsto \boldsymbol{z}_i$
Parameters	$\boldsymbol{\theta}$
Loss	$L(\boldsymbol{\theta}) = \sum_i \left\| f(\mathbf{x}_i; \boldsymbol{\theta}) - \mathbf{z}_i \right\|^2$
Update	$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \alpha \cdot \nabla L(\boldsymbol{\theta}^{(t)}) + \beta \cdot (\boldsymbol{\theta}^{(t)} - \boldsymbol{\theta}^{(t-1)})$

8.1 Linear regression

Examples	$0 \mapsto 26$
	$10 \mapsto 31$
	$20 \mapsto 40$
Parameters	$\boldsymbol{\theta} = (\theta_0, \theta_1)$
Line	$f(x; \theta_0, \theta_1) = \theta_0 + \theta_1 x$

8.2 Perceptron

Examples	$(-.9, -.9) \mapsto +.9$
	$(-.9, +.9) \mapsto +.9$
	$(+.9, -.9) \mapsto +.9$
	$(+.9, +.9) \mapsto -.9$
Parameters	$\boldsymbol{\theta} = (\theta_0, \theta_1, \theta_2)$
Neuron	$f(x, y; \theta_0, \theta_1, \theta_2) = \sigma(\theta_0 + \theta_1 x + \theta_2 y)$
Sigmoid	$\sigma(r) = \dfrac{2}{1 + e^{-r}} - 1$



8.3 Network

Examples

$$\begin{aligned}(-.9, -.9) &\mapsto -.9\\(-.9, +.9) &\mapsto +.9\\(+.9, -.9) &\mapsto +.9\\(+.9, +.9) &\mapsto -.9\end{aligned}$$

Parameters

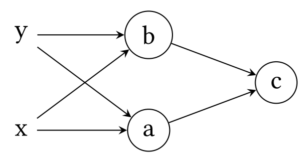
$$\begin{aligned}\boldsymbol{\theta} &= (\mathbf{a}, \mathbf{b}, \mathbf{c})\\\mathbf{a} &= (a_0, a_1, a_2)\\\mathbf{b} &= (b_0, b_1, b_2)\\\mathbf{c} &= (c_0, c_1, c_2)\end{aligned}$$

Network

$$f(x, y; \mathbf{a}, \mathbf{b}, \mathbf{c}) = g\big(g(x, y; \mathbf{a}), g(x, y; \mathbf{b}); \mathbf{c}\big)$$

Neuron

$$g(x, y; a_0, a_1, a_2) = \sigma(a_0 + a_1 x + a_2 y)$$



9 AUTOMATIC DIFFERENTIATION

[Krawiec et al. 2022]

REFERENCES

Fischer, Sebastian, Oleg Kiselyov, and Chung-chieh Shan. 2011. Purely functional lazy nondeterministic programming. *Journal of Functional Programming* 21(4–5):413–465.

Hutton, Graham, and Erik Meijer. 1998. Monadic parsing in Haskell. *Journal of Functional Programming* 8(4):437–444.

Krawiec, Faustyna, Simon Peyton Jones, Neel Krishnaswami, Tom Ellis, Richard A. Eisenberg, and Andrew Fitzgibbon. 2022. Provably correct, asymptotically efficient, higher-order reverse-mode automatic differentiation. *Proceedings of the ACM on Programming Languages* 6(POPL):48:1–48:30.

Liang, Sheng, Paul Hudak, and Mark Jones. 1995. Monad transformers and modular interpreters. In *POPL '95: Conference record of the annual ACM symposium on principles of programming languages*, 333–343. New York: ACM Press.

Peyton Jones, Simon L. 2001. Tackling the awkward squad: Monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In *Engineering theories of software construction*, ed. Tony Hoare, Manfred Broy, and Ralf Steinbruggen, 47–96. NATO Science Series: Computer and Systems Sciences 180, Amsterdam: IOS Press. Presented at the 2000 Marktoberdorf Summer School.

Ramsey, Norman, and Avi Pfeffer. 2002. Stochastic lambda calculus and monads of probability distributions. In *POPL '02: Conference record of the annual ACM symposium on principles of programming languages*, 154–165. New York: ACM Press.

Wadler, Philip L. 1995. Monads for functional programming. In *Advanced functional programming: 1st international spring school on advanced functional programming techniques*, ed. Johan Jeuring and Erik Meijer, 24–52. Lecture Notes in Computer Science 925, Berlin: Springer.

Wadler, Philip L., and Stephen Blott. 1989. How to make *ad-hoc* polymorphism less *ad hoc*. In *POPL '89: Conference record of the annual ACM symposium on principles of programming languages*, 60–76. New York: ACM Press.