Monad and side effects

單中杰

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.

Property-based testing [Claessen and Hughes 2000]

Side effects

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

Use each side effect to write some programs, including an interpreter for a simple language. We want to reuse the same interpreter code for a variety of side effects.

2 State (the OG side effect)

- $\mathit{state} \to (\mathit{value}, \mathit{state})$
- Local vs global state (example: union-find)

3 Exception, Maybe

- $\mathit{error} + \mathit{value}$
- Early exit from loop/recursion

4 Nondeterminism

- Set/multiset/list/pointed-set/distribution of values
- Backtracking/tabling search

5 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:

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

6 Type classes

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

- Passed implicitly
- Constructed automatically during type inference

6.1 Eq

Default method implementations

$$\begin{array}{l}
\textbf{class}\ \mathit{Eq}\ a\ \textbf{where} \\
\quad (==) :: a \to a \to \mathit{Bool} \\
\quad (/=) :: a \to a \to \mathit{Bool} \\
\quad x \mathrel{/=} y = \mathit{not}\ (x == y) \\
\quad x == y = \mathit{not}\ (x \mathrel{/=} y)
\end{array}$$

$$\begin{array}{l}
\textbf{instance}\ \mathit{Eq}\ \mathit{Int}\ \ \textbf{where}\quad \text{-- ...} \\
\textbf{instance}\ \mathit{Eq}\ \mathit{Char}\ \textbf{where}\quad \text{-- ...}
\end{array}$$

Instance contexts (constraints)

$$\begin{array}{l}
\textbf{instance}\ (\mathit{Eq}\ a, \mathit{Eq}\ b) \Rightarrow \mathit{Eq}\ (a, b)\ \textbf{where} \\
\quad (a1, b1) == (a2, b2) = a1 == a2\ \&\&\ b1 == b2
\end{array}$$

$$\begin{array}{l}
\textbf{instance}\ (\mathit{Eq}\ a) \Rightarrow \mathit{Eq}\ [a]\ \textbf{where} \\
\quad \begin{array}{lll}
[\,] & == [\,] & = \mathit{True} \\
(x : xs) & == (y : ys) & = x == y\ \&\&\ xs == ys \\
_ & == _ & = \mathit{False}
\end{array}
\end{array}$$

Type-signature contexts (constraints)

$$\mathit{elem} :: (\mathit{Eq}\ a) \Rightarrow a \to [a] \to \mathit{Bool}$$

6.2 Ord

Class contexts (superclasses)

$$
\begin{array}{l}
\textbf{class}\ (Eq\ a) \Rightarrow Ord\ a\ \textbf{where} \\
\quad (<), (\leqslant), (>), (\geqslant) :: a \to a \to Bool \\
\quad x < y\ = not\ (x == y)\ \&\&\ \quad\ \ (x \leqslant y) \\
\quad x > y\ = not\ (x == y)\ \&\&\ not\ (x \leqslant y) \\
\quad x \geqslant y = \quad\ \ (x == y)\ \ \|\ \ not\ (x \leqslant y) \\
\quad \text{-- ...}
\end{array}
$$

6.3 Show

$$
\begin{array}{l}
\textbf{class}\ Show\ a\ \textbf{where} \\
\quad show :: a \to String \\
\quad \text{-- ...}
\end{array}
$$

6.4 Monad

Refined class hierarchy

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

Easy superclass implementations

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

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

7 Imperative programming

Input, Output

- $input \to value$
- $(value, output)$
- Interactivity: inputs and outputs that depend on each other

Control

- GO TO
- Continuations
- Measures?! Everything?!?! [Filinski 1994]

7.1 Do notation

Imperative intuition. For example, monad laws:

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

7.2 Polymorphism across monads

Action combinators, useful for all monads

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

$\textbf{import}\ Control.Monad$

$$
\begin{array}{ll}
replicateM & :: (Monad\ m) \Rightarrow Int \to m\ a \to m\ [a] \\
replicateM\_ & :: (Monad\ m) \Rightarrow Int \to m\ a \to m\ () \\
filterM & :: (Monad\ m) \Rightarrow (a \to m\ Bool) \to [a] \to m\ [a] \\
foldM & :: (Monad\ m) \Rightarrow (b \to a \to m\ b) \to b \to [a] \to m\ b
\end{array}
$$

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

8 Combining side effects

8.1 State and IO

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

import *Control.Monad.Trans.State*

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

8.2 State and exception

Different combinations have different meanings

import *Control.Monad.Trans.Maybe*

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

Stack of memory regions

8.3 State and nondeterminism

Different combinations have different meanings

import *Control.Monad.Trans.List*

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

Lazy evaluation in state [Fischer et al. 2011]

8.4 Monad transformers

[Liang et al. 1995]

import *Control.Monad.Trans.Class*
class *MonadTrans t* **where**
  $lift :: (Monad\ m) \Rightarrow m\ a \rightarrow t\ m\ a$

Monad transformer laws

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

Operations need to be lifted too

9 Parsing

[Hutton and Meijer 1998]

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

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

instance *Monad Parser*
$empty :: Parser\ a$
$(\langle|\rangle) :: Parser\ a \rightarrow Parser\ a \rightarrow Parser\ a$
$item :: Parser\ Char$

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

$$
\begin{aligned}
&empty\ \langle|\rangle\ p &&= p\\
&p\ \langle|\rangle\ empty &&= p\\
&p\ \langle|\rangle\ (q\ \langle|\rangle\ r) &&= (p\ \langle|\rangle\ q)\ \langle|\rangle\ r\\
&empty \ggg\!= f &&= empty\\
&p \ggg\!= \backslash\_ \rightarrow empty &&= empty\\
&(p\ \langle|\rangle\ q) \ggg\!= f &&= (p \ggg\!= f)\ \langle|\rangle\ (q \ggg\!= f)\\
&p \ggg\!= (\lambda a \rightarrow f\ a\ \langle|\rangle\ g\ a) &&= (p \ggg\!= f)\ \langle|\rangle\ (p \ggg\!= g)
\end{aligned}
$$

10 Probability

[Ramsey and Pfeffer 2002]

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

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

instance *Monad Dist*
$empty :: Dist\ a$
$(\langle|\rangle) :: Dist\ a \rightarrow Dist\ a \rightarrow Dist\ a$
$factor :: Double \rightarrow Dist\ ()$

11 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 $\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \alpha \cdot \nabla L(\boldsymbol{\theta}^{(t)}) + \beta \cdot (\boldsymbol{\theta}^{(t)} - \boldsymbol{\theta}^{(t-1)})$

11.1 Linear regression

Examples
$$0 \mapsto 26$$
$$10 \mapsto 31$$
$$20 \mapsto 40$$

Parameters $\boldsymbol{\theta} = (\theta_0, \theta_1)$

Line $f(x; \theta_0, \theta_1) = \theta_0 + \theta_1 x$

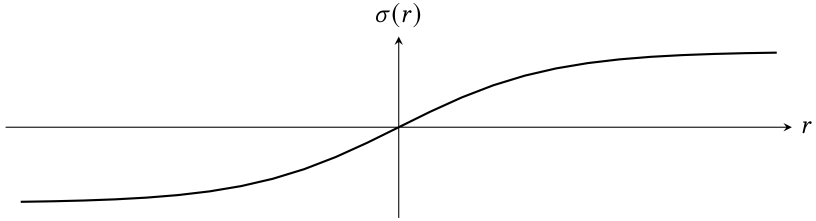
11.2 Perceptron

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 $\sigma(r) = \dfrac{2}{1 + e^{-r}} - 1$



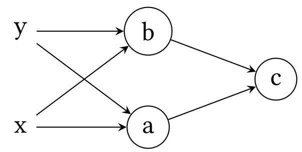
11.3 Network

Examples
$$(-.9, -.9) \mapsto -.9$$
$$(-.9, +.9) \mapsto +.9$$
$$(+.9, -.9) \mapsto +.9$$
$$(+.9, +.9) \mapsto -.9$$

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

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



12 Automatic differentiation

[Krawiec et al. 2022]

$$
\begin{aligned}
&\mathit{runIdentity} \circ \mathit{eval}\ e\ \ :: \mathit{M.Map\ Name\ Double} \qquad\qquad \rightarrow \mathit{Double}\\
&\mathit{runIdentity} \circ \mathit{eval2}\ e :: \mathit{M.Map\ Name}\ (\mathit{Double}, \mathit{Double}) \rightarrow (\mathit{Double}, \mathit{Double})
\end{aligned}
$$

$$
\begin{aligned}
&(\mathit{runIdentity} \circ \mathit{eval2}\ e)\ (\mathit{env})\\
&== ((\mathit{runIdentity} \circ \mathit{eval}\ e)\ (\mathit{fmap\ fst\ env})\\
&\quad , \nabla(\mathit{runIdentity} \circ \mathit{eval}\ e)\ (\mathit{fmap\ fst\ env}) \bullet (\mathit{fmap\ snd\ env}))\\
&\quad \textbf{where}\ (\bullet) :: \mathit{M.Map\ Name\ Double} \rightarrow \mathit{M.Map\ Name\ Double} \rightarrow \mathit{Double}\\
&\qquad\quad \mathit{um} \bullet \mathit{vm} = \mathit{M.foldr}\ (+)\ 0\ (\mathit{M.intersectionWith}\ (\times)\ \mathit{um\ vm})
\end{aligned}
$$

$$
\mathit{runIdentity} \circ \mathit{eval3}\ e :: \mathit{M.Map\ Name}\ (\mathit{Double}, \mathit{Delta}) \quad \rightarrow (\mathit{Double}, \mathit{Delta})
$$

$$
\begin{aligned}
&(\mathit{runIdentity} \circ \mathit{eval3}\ e)\ (\mathit{env})\\
&== ((\mathit{runIdentity} \circ \mathit{eval}\ e)\ (\mathit{fmap\ fst\ env})\\
&\quad , \nabla(\mathit{runIdentity} \circ \mathit{eval}\ e)\ (\mathit{fmap\ fst\ env}) \bullet (\mathit{fmap\ snd\ env}))\\
&\quad \textbf{where}\ (\bullet) :: \mathit{M.Map\ Name\ Double} \rightarrow \mathit{M.Map\ Name\ Delta} \rightarrow \mathit{Delta}\\
&\qquad\quad \mathit{um} \bullet \mathit{vm} = \mathit{M.foldr\ dAdd\ M.empty}\ (\mathit{M.intersectionWith\ dScale\ um\ vm})
\end{aligned}
$$

$$
\mathit{runDelta} \circ \mathit{eval4}\ e \quad :: \mathit{M.Map\ Name}\ (\mathit{Double}, \mathit{Delta}) \quad \rightarrow (\mathit{Double}, \mathit{Delta})
$$

$$
\begin{aligned}
&(\mathit{runDelta} \circ \mathit{eval4}\ e)\ (\mathit{env})\\
&== ((\mathit{runIdentity} \circ \mathit{eval}\ e)\ (\mathit{fmap\ fst\ env})\\
&\quad , \nabla(\mathit{runIdentity} \circ \mathit{eval}\ e)\ (\mathit{fmap\ fst\ env}) \bullet (\mathit{fmap\ snd\ env}))\\
&\quad \textbf{where}\ (\bullet) :: \mathit{M.Map\ Name\ Double} \rightarrow \mathit{M.Map\ Name\ Delta} \rightarrow \mathit{Delta}\\
&\qquad\quad \mathit{um} \bullet \mathit{vm} = \mathit{M.foldr\ DAdd\ Zero}\ (\mathit{M.intersectionWith\ DScale\ um\ vm})
\end{aligned}
$$

References

Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *ICFP '00: Proceedings of the ACM International Conference on Functional Programming* (Montréal, Québec, Canada) *(ACM SIGPLAN Notices, Vol. 35(9))*. ACM Press, New York, 268–279. https://doi.org/10.1145/351240.351266

Andrzej Filinski. 1994. Representing Monads. In *POPL '94: Conference Record of the Annual ACM Symposium on Principles of Programming Languages* (Portland, OR). ACM Press, New York, 446–457. http://hjemmesider.diku.dk/~andrzej/papers/RM-abstract.html

Sebastian Fischer, Oleg Kiselyov, and Chung-chieh Shan. 2011. Purely Functional Lazy Nondeterministic Programming. *Journal of Functional Programming* 21, 4–5 (2011), 413–465. https://doi.org/10.1017/S0956796811000189

Graham Hutton and Erik Meijer. 1998. Monadic Parsing in Haskell. *Journal of Functional Programming* 8, 4 (July 1998), 437–444. https://doi.org/10.1017/S0956796898003050

Faustyna Krawiec, 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 (Jan. 2022), 48:1–48:30. https://doi.org/10.1145/3498710

Sheng Liang, 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* (San Francisco, CA). ACM Press, New York, 333–343. https://doi.org/10.1145/199448.199528

Simon L. Peyton Jones. 2001. Tackling the Awkward Squad: Monadic Input/Output, Concurrency, Exceptions, and Foreign-Language Calls in Haskell. In *Engineering Theories of Software Construction (NATO Science Series: Computer and Systems Sciences, 180)*, Tony Hoare, Manfred Broy, and Ralf Steinbruggen (Eds.). IOS Press, Amsterdam, 47–96. https://www.microsoft.com/en-us/research/publication/tackling-awkward-squad-monadic-inputoutput-concurrency-exceptions-foreign-language-calls-haskell/ Presented at the 2000 Marktoberdorf Summer School..

Norman Ramsey 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* (Portland, OR). ACM Press, New York, 154–165. https://www.cs.tufts.edu/~nr/pubs/pmonad-abstract.html

Philip L. Wadler. 1995. Monads for Functional Programming. In *Advanced Functional Programming: 1st International Spring School on Advanced Functional Programming Techniques*, Johan Jeuring and Erik Meijer (Eds.). Number 925 in Lecture Notes in Computer Science. Springer, Berlin, 24–52. https://homepages.inf.ed.ac.uk/wadler/topics/monads.html#marktoberdorf

Philip L. Wadler 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* (Austin). ACM Press, New York, 60–76. https://doi.org/10.1145/75277.75283