

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)
 - $state \rightarrow (value, state)$
 - Local vs global state
- Exception, Maybe
 - $error + value$
- Nondeterminism
 - set/multiset/list/pointed-set/distribution of values
- Input, Output
 - $input \rightarrow value$
 - $(value, output)$
 - Interactivity
- ...

3 MONADS

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

$return :: \forall a. a \rightarrow M a$ -- unit, eta
 $(\gg) :: \forall a b. M a \rightarrow (a \rightarrow M b) \rightarrow M b$ -- bind, star

Monad laws:

$return a \gg k = k a$
 $m \gg return = m$
 $m \gg (\lambda a \rightarrow k a \gg l) = (m \gg k) \gg l$

Alternative definition with three operations:

$return :: \forall a. a \rightarrow M a$ -- unit, eta
 $fmap :: \forall a b. (a \rightarrow b) \rightarrow M a \rightarrow M b$ -- functoriality
 $join :: \forall a. M (M a) \rightarrow M a$ -- mu

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**

$(==) :: a \rightarrow a \rightarrow \text{Bool}$

$(/=) :: a \rightarrow a \rightarrow \text{Bool}$

$x /= y = \text{not } (x == y)$

$x == y = \text{not } (x /= y)$

instance *Eq Int* **where** -- ...

instance *Eq Char* **where** -- ...

instance (*Eq a*) \Rightarrow *Eq [a]* **where**

$[] == [] = \text{True}$

$(x : xs) == (y : ys) = x == y \ \&\& \ xs == ys$

$_ == _ = \text{False}$

instance (*Eq a*, *Eq b*) \Rightarrow *Eq (a, b)* **where**

$(a1, b1) == (a2, b2) = a1 == a2 \ \&\& \ b1 == b2$

Instance contexts (constraints)

Type-signature contexts (constraints)

$elem :: (Eq a) \Rightarrow a \rightarrow [a] \rightarrow \text{Bool}$

4.2 Ord

Class contexts (superclasses)

class (*Eq a*) \Rightarrow *Ord a* **where**

$(<), (\leq), (>), (\geq) :: a \rightarrow a \rightarrow \text{Bool}$

$x < y = \text{not } (x == y) \ \&\& \ (x \leq y)$

$x > y = \text{not } (x == y) \ \&\& \ \text{not } (x \leq y)$

$x \geq y = (x == y) \ \parallel \ \text{not } (x \leq y)$

-- ...

4.3 Show

class *Show a* **where**

$show :: a \rightarrow \text{String}$

-- ...

4.4 Monad

Refined class hierarchy

```

class Functor m where
  fmap :: (a → b) → m a → m b
class (Functor m) ⇒ Applicative m where
  pure  :: a → m a
  (<*>) :: m (a → b) → m a → m b
class (Applicative m) ⇒ Monad m where
  return :: a → m a
  (>>=)  :: m a → (a → m b) → m b

```

Easy superclass implementations

```

newtype State s a = State { runState :: s → (a, s) }
instance Functor (State s) where fmap = liftM
instance Applicative (State s) where pure = return; (<*>) = ap
instance Monad (State s) where
  return a = State (λs → (a, s))
  m >>= k = State (λs → let (a, s') = runState m s
                        in runState (k a) s')

```

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

5 DO NOTATION

Imperative intuition. For example, monad laws:

```

do { x ← return a; k x }      = k a
do { x ← m; return x }        = m
do { a ← m; do { b ← k a; l b } } = do { b ← do { a ← m; k a }; l b }

```

6 POLYMORPHISM ACROSS MONADS

Action combinators, useful for all monads

```

traverse  :: (Monad m) ⇒ (a → m b) → [a] → m [b]
traverse_ :: (Monad m) ⇒ (a → m b) → [a] → m ()
sequence  :: (Monad m) ⇒ [m a] → m [a]
sequence_ :: (Monad m) ⇒ [m a] → m ()

```

```
import Control.Monad
```

```

replicateM :: (Monad m) ⇒ Int → m a → m [a]
replicateM_ :: (Monad m) ⇒ Int → m a → m ()
filterM    :: (Monad m) ⇒ (a → m Bool) → [a] → m [a]
foldM      :: (Monad m) ⇒ (b → a → m b) → b → [a] → m b

```

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]

```

class MonadTrans t where
  lift :: (Monad m) ⇒ m a → t m a

```

Monad transformer laws

$\text{lift } (\text{return } a) = \text{return } a$
 $\text{lift } (m \ggg k) = \text{lift } m \ggg (\text{lift } \circ k)$

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

$\text{StateT } s \text{ IO } a = s \rightarrow \text{IO } (a, s)$

7.2 State and exception

Different combinations have different meanings

$\text{StateT } s \text{ Maybe } a = s \rightarrow \text{Maybe } (a, s)$
 $\text{MaybeT } (\text{State } s) a = s \rightarrow (\text{Maybe } a, s)$
 $\text{StateT } s (\text{MaybeT } (\text{State } t)) a = s \rightarrow t \rightarrow (\text{Maybe } (a, s), t)$

Stack of memory regions

7.3 State and nondeterminism

Different combinations have different meanings

$\text{StateT } s \text{ [] } a = s \rightarrow [(a, s)]$
 $\text{ListT } (\text{State } s) a = s \rightarrow ([a], s)$

Lazy evaluation in state [Fischer et al. 2011]

7.4 Parsing

[Hutton and Meijer 1998]

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

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

instance *Monad Parser*

empty :: *Parser a*

$(\langle \rangle) :: \text{Parser } a \rightarrow \text{Parser } a \rightarrow \text{Parser } a$

item :: *Parser Char*

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

$\text{empty } \langle \rangle p = p$
 $p \langle \rangle \text{empty} = p$
 $p \langle \rangle (q \langle \rangle r) = (p \langle \rangle q) \langle \rangle r$
 $\text{empty } \ggg f = \text{empty}$
 $p \ggg _ \rightarrow \text{empty} = \text{empty}$
 $(p \langle \rangle q) \ggg f = (p \ggg f) \langle \rangle (q \ggg f)$
 $p \ggg (\lambda a \rightarrow f a \langle \rangle g a) = (p \ggg f) \langle \rangle (p \ggg g)$

7.5 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

(⟨|⟩) :: Dist a → Dist a → Dist a

factor :: Double → Dist ()

8 NEURAL NETS

Examples

$$\mathbf{x}_i \mapsto \mathbf{z}_i$$

Parameters

$$\boldsymbol{\theta}$$

Loss

$$L(\boldsymbol{\theta}) = \sum_i \|f(\mathbf{x}_i; \boldsymbol{\theta}) - \mathbf{z}_i\|^2$$

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

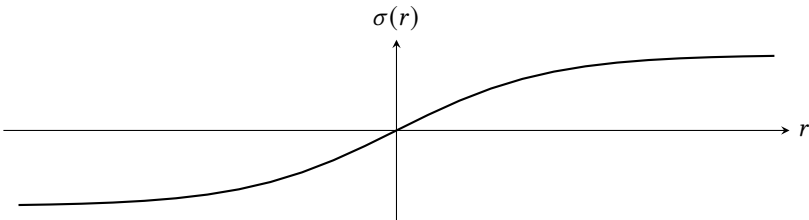
$$\boldsymbol{\theta} = (\theta_0, \theta_1, \theta_2)$$

Neuron

$$f(x, y; \theta_0, \theta_1, \theta_2) = \sigma(\theta_0 + \theta_1 x + \theta_2 y)$$

Sigmoid

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



8.3 Network

Examples

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

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

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

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

Parameters

$$\theta = (\mathbf{a}, \mathbf{b}, \mathbf{c})$$

$$\mathbf{a} = (a_0, a_1, a_2)$$

$$\mathbf{b} = (b_0, b_1, b_2)$$

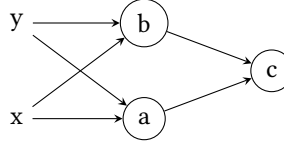
$$\mathbf{c} = (c_0, c_1, c_2)$$

Network

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

Neuron

$$g(x, y; a_0, a_1, a_2) = \sigma(a_0 + a_1x + a_2y)$$



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.