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

1.1 Pure recursion

`Tree-1.hs`

For modular reuse, abstract from similarities over differences: *sumTree* vs *productTree*

1.2 Interpreter

`Arith-1.hs`

Challenge: local variable binding `Arith-2.hs`

2 STATE

Certain programming tasks make us intuitively reach for side effects.

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

2.1 Accumulator passing

`TreeState-1.hs`

```
result := 0
sumTree (Leaf n)        = result := result + n;
                          result
sumTree (Branch t1 t2) = sumTree t1;
                          sumTree t2
```

2.2 State threading

`TreeState-2.hs`

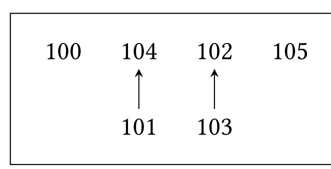
```
next := 0
relabel (Leaf _)        = next := next + 1;
                          Leaf next
relabel (Branch t1 t2) = Branch (relabel t1) (relabel t2)

seen := S.empty
unique (Leaf n)         = if S.member n seen then False
                          else seen := S.insert n seen; True
unique (Branch t1 t2)  = unique t1 && unique t2
```

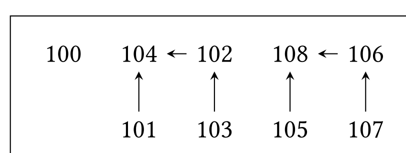
2.3 Local vs global state

`UnionFind-1.hs`

```
testState :: State
testState = M.fromList [(Key 100, Root 0 "A")
                       , (Key 101, Link (Key 104))
                       , (Key 102, Root 1 "C")
                       , (Key 103, Link (Key 102))
                       , (Key 104, Root 1 "E")
                       , (Key 105, Root 0 "F")]
```




```
testState' :: State
testState' = M.fromList [(Key 100, Root 0 "A")
                        , (Key 101, Link (Key 104))
                        , (Key 102, Link (Key 104))
                        , (Key 103, Link (Key 102))
                        , (Key 104, Root 2 "E")
                        , (Key 105, Link (Key 108))
                        , (Key 106, Link (Key 108))
                        , (Key 107, Link (Key 106))
                        , (Key 108, Root 2 "I")]
```



Pointers, references, file system

2.4 Interpreter

`ArithState-1.hs`

Challenge: Memory allocation `ArithState-2.hs`

```
data Expr = Lit Int | Add Expr Expr | Mul Expr Expr
          | New Expr | Get Expr | Put Expr Expr
type State = [Int]
```

How can this be useful?

3 EXCEPTION

3.1 Tree

`TreeMaybe-1.hs`

3.2 Interpreter

`ArithMaybe-1.hs`

4 NONDETERMINISM

4.1 Tree

`TreeNondet-1.hs`

$$
\begin{array}{l}
\mathit{blackjack}' :: \mathit{Tree} \rightarrow \mathit{Int} \rightarrow [\,\mathit{Int}\,] \\
\mathit{blackjack}'\ (\mathit{Leaf}\ n)\qquad\quad \mathit{total} = \textbf{if}\ \mathit{total} + n > 21\ \textbf{then}\ \mathit{total} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{else}\ \mathit{amb}\ [\,\mathit{total}, \mathit{total} + n\,] \\
\mathit{blackjack}'\ (\mathit{Branch}\ \mathit{t1}\ \mathit{t2})\ \mathit{total} = \mathit{blackjack}'\ \mathit{t2}\ (\mathit{blackjack}'\ \mathit{t1}\ \mathit{total})
\end{array}
$$

4.2 SEND + MORE = MONEY

`Crypta-1.hs`

Loop bodies are continuation functions.

Prepone checking to avoid futile generation.

`Crypta-2.hs`

Use state to remember letters whose digits have been chosen.

Generalize to TO + GO = OUT.

4.3 Interpreter

`ArithNondet-1.hs`

5 MONADS

Abstract from *Expr* interpreter. [Wadler 1995]

6 TYPE CLASSES

Examples: *Eq*, *Ord*, *Show*. [Wadler and Blott 1989]

Monad class, inheriting from *Applicative*, inheriting from *Functor*.

`ArithMonad-1.hs` `ArithMonad-2.hs` `ArithMonad-3.hs`

7 IMPERATIVE PROGRAMMING

`ArithIO-1.hs`

How (and in what monad) to interpret *Input* and *Output* is an open-ended question.

> A value of type *IO a* is an "action" that, when performed, may do some input/output, before delivering a value of type *a*.
>
> $\textbf{type}\ \mathit{IO}\ a = \mathit{World} \rightarrow (a, \mathit{World})$

Execution by *monad laws* and labeled transitions ([Peyton Jones 2001, Figure 3])

Translating impure programs to monadic form

7.1 Do notation

`ArithDo-1.hs` `ArithDo-2.hs` `ArithDo-3.hs` `ArithDo-4.hs`

7.2 Polymorphism across monads

`Traverse-1.hs`

8 COMBINING SIDE EFFECTS

8.1 State and IO

`StateIO-1.hs` `StateIO-2.hs`

8.2 State and exception

`StateMaybe-1.hs` `StateMaybe-2.hs`

8.3 State and nondeterminism

`StateNondet-1.hs` `StateNondet-2.hs`
[Fischer et al. 2011]

8.4 Monad transformers

`StateIO-3.hs` `StateMaybe-3.hs` `StateMaybe-4.hs` `StateNondet-3.hs` `StateNondet-4.hs` [Liang et al. 1995]

$$\begin{aligned}
&\mathit{StateT}\ s\ \mathit{IO} &&= s \to \mathit{IO}\ (a, s)\\
&\mathit{StateT}\ s\ \mathit{Maybe}\ a &&= s \to \mathit{Maybe}\ (a, s)\\
&\mathit{MaybeT}\ (\mathit{State}\ s)\ a &&= s \to (\mathit{Maybe}\ a, s)\\
&\mathit{StateT}\ s\ (\mathit{MaybeT}\ (\mathit{State}\ t))\ a &&= s \to t \to (\mathit{Maybe}\ (a, s), t)\\
&\mathit{StateT}\ s\ [\,]\ a &&= s \to [\,(a, s)\,]\\
&\mathit{ListT}\ (\mathit{State}\ s)\ a &&= s \to ([\,a\,], s) \quad \text{-- for profiling search?}
\end{aligned}$$

Lifting operations is ad hoc

9 PARSING

[Hutton and Meijer 1998]

type *Parser* = *StateT String* [ ]

Prove monad laws
Disprove left distributivity for +++
apply expr `" 1 - 2 * 3 + 4 "` with (+++) = (⟨|⟩)

10 PROBABILITY

[Ramsey and Pfeffer 2002]

type *Dist* = *WriterT* (*Product Double*) [ ]

11 NEURAL NETS

12 AUTOMATIC DIFFERENTIATION

[Krawiec et al. 2022]

```
import Test.QuickCheck (quickCheckAll, (==>), Arbitrary (arbitrary), frequency)
import Control.Monad.Identity
import System.Random (getStdRandom, randomR)
import Numeric (showGFloat)
import Control.Monad.Trans.State
import qualified Data.Map as M

data Expr = Lit Double
          | Add Expr Expr
          | Mul Expr Expr
          | Pow Expr Double
          | Exp Expr
          | Var Name
          | Let Name Expr Expr
  deriving (Eq, Show)
type Name = String

freeVars :: Expr → M.Map Name ()
freeVars (Lit _)       = M.empty
freeVars (Var n)       = M.singleton n ()
freeVars (Add e1 e2)   = M.union (freeVars e1) (freeVars e2)
freeVars (Mul e1 e2)   = M.union (freeVars e1) (freeVars e2)
freeVars (Pow e _)     = freeVars e
freeVars (Exp e)       = freeVars e
freeVars (Let n rhs e) = M.union (freeVars rhs) (M.delete n (freeVars e))

sigmoid :: Expr → Expr
sigmoid e = Add (Mul (Lit 2) (Pow (Add (Lit 1) (Exp (Mul (Lit (−1)) e)))
                                  (−1)))
                (Lit (−1))
```

```
eval :: (Monad m) ⇒ Expr → M.Map Name Double → m Double
prop_eval_Let                 = runIdentity (eval (Let "x" (Add (Lit 3) (Lit (−1)))
                                                   (Mul (Var "x") (Var "x")))
                                   M.empty)
                                == 4
prop_eval_square v            = abs (runIdentity (eval (Pow (Lit v) 2) M.empty) −
                                     runIdentity (eval (Mul (Lit v) (Lit v)) M.empty))
                                < 0.001
prop_eval_sigmoid0       env = 0 == runIdentity (eval (sigmoid (Lit 0)) env)
prop_eval_sigmoid1 v     env = let sv  = runIdentity (eval (sigmoid (Lit v)) env)
                               in − 1 ⩽ sv && sv ⩽ 1
prop_eval_sigmoid2 v1 v2 env = v1 < v2 ==>
                               let sv1 = runIdentity (eval (sigmoid (Lit v1)) env)
                                   sv2 = runIdentity (eval (sigmoid (Lit v2)) env)
                               in sv1 < sv2 || sv1 == sv2 && (v2 − v1 < 0.1 || v2 < −30 || v1 > 30)
eval (Lit v)       _   = return v
eval (Add e1 e2)   env = do v1 ← eval e1 env
                            v2 ← eval e2 env
                            return (v1 + v2)
eval (Mul e1 e2)   env = do v1 ← eval e1 env
                            v2 ← eval e2 env
                            return (v1 × v2)
eval (Var n)       env = do return (env M.! n)
eval (Let n rhs e) env = do v ← eval rhs env
                            eval e (M.insert n v env)
```

[Claessen and Hughes 2000]

```
instance Arbitrary Expr where
  arbitrary = frequency [ (6, liftM   Lit   arbitrary)
                        , (1, liftM2 Add arbitrary arbitrary)
                        , (1, liftM2 Mul arbitrary arbitrary)
                        , (1, liftM2 Pow arbitrary arbitrary)
                        , (1, liftM   Exp  arbitrary) ]

eval₂ :: (Monad m) ⇒ Expr → M.Map Name (Double, Double) → m (Double, Double)
prop_eval2 e            = let v0     = runIdentity (eval   e M.empty)
                              (v, _) = runIdentity (eval₂ e M.empty)
                          in v0 == v || isNaN v0 && isNaN v
prop_eval2_sigmoid u = runIdentity (eval₂ (sigmoid (Var "x"))
                                          (M.singleton "x" (0, u)))
                       == (0, u / 2)

eval₂ (Lit v)         _   = return (v, 0)
eval₂ (Add e₁ e₂)   env = do (v₁, u₁) ← eval₂ e₁ env
                             (v₂, u₂) ← eval₂ e₂ env
                             return (v₁ + v₂, u₁ + u₂)
eval₂ (Mul e₁ e₂)   env = do (v₁, u₁) ← eval₂ e₁ env
                             (v₂, u₂) ← eval₂ e₂ env
                             return (v₁ × v₂, v₁ × u₂ + v₂ × u₁)
eval₂ (Var n)       env = do return (env M.! n)
eval₂ (Let n rhs e) env = do v ← eval₂ rhs env
                             eval₂ e (M.insert n v env)

sum_ :: [Expr] → Expr
sum_ = foldl Add (Lit 0)

perceptron :: Expr → (Expr, Expr) → (Expr, Expr) → Expr
perceptron a0 (a1, x) (a2, y) = sigmoid (sum_ [a0, Mul a1 x, Mul a2 y])

network :: Expr
network = Let "a" (perceptron (Var "a0") (Var "a1", Var "x") (Var "a2", Var "y"))
            (Let "b" (perceptron (Var "b0") (Var "b1", Var "x") (Var "b2", Var "y"))
              (Let  "c" (perceptron (Var "c0") (Var "c1", Var "a") (Var "c2", Var "b"))
                (Var "c")))

loss :: Expr
loss = sum_  [ Pow  (Add (Let "x" (Lit x) (Let "y" (Lit y) network))
                         (Lit (negate expect)))
                    2
             | (x, y, expect) ← [ (−0.9, −0.9, −0.9)
                                , (−0.9,   0.9,   0.9)
                                , (  0.9, −0.9,   0.9)
                                , (  0.9,   0.9, −0.9) ] ]
```

```
type Params = M.Map Name Double
randomParams :: IO Params
randomParams = traverse (λ() → getStdRandom (randomR (−2, 2)))
                        (freeVars loss)
stepParams :: Params → Params
stepParams params =
  let stepParam n v =
    let dualize nn vv = (vv, if n == nn then 1 else 0)
    in v − 0.1 × snd (runIdentity (eval₂ loss (M.mapWithKey dualize params)))
  in M.mapWithKey stepParam params
showF :: Double → String
showF v = showGFloat (Just 3) v ""
optimize :: Params → IO ()
optimize params = do
  mapM_ (λv → putStr (showF v ++ "  ")) params
  putStrLn ("=>  " ++ showF (runIdentity (eval loss params)))
  optimize (iterate stepParams params !! 1000)
  randomParams >>= optimize
type Params = M.Map Name Inertia
data Inertia = Inertia Double Double
  deriving (Eq, Show)

type Delta = M.Map Name Double
eval₃ :: (Monad m) ⇒ Expr → M.Map Name (Double, Delta) → m (Double, Delta)
prop_eval3 e          = let v0     = runIdentity (eval  e M.empty)
                              (v, _) = runIdentity (eval₃ e M.empty)
                          in v0 == v || isNaN v0 && isNaN v
prop_eval3_sigmoid u = runIdentity (eval₃ (sigmoid (Var "x"))
                                           (M.singleton "x" (0, u)))
                         == (0, M.map (/2) u)
dAdd :: Delta → Delta → Delta
dAdd = M.unionWith (+)
dScale :: Double → Delta → Delta
dScale v = M.map (v×)
eval₃ (Lit v)       _   = return (v, M.empty)
eval₃ (Add e₁ e₂)  env = do (v₁, u₁) ← eval₃ e₁ env
                             (v₂, u₂) ← eval₃ e₂ env
                             return (v₁ + v₂, dAdd u₁ u₂)
eval₃ (Mul e₁ e₂)  env = do (v₁, u₁) ← eval₃ e₁ env
                             (v₂, u₂) ← eval₃ e₂ env
                             return (v₁ × v₂, dAdd (dScale v₁ u₂) (dScale v₂ u₁))
eval₃ (Var n)       env = do return (env M.! n)
eval₃ (Let n rhs e) env = do v ← eval₃ rhs env
                             eval₃ e (M.insert n v env)
```

```
data Delta = Zero
           | DAdd Delta Delta
           | DScale Double Delta
           | DVar DeltaId
           | DLet DeltaId Delta Delta
  deriving (Eq, Show)
data DeltaId = Name Name | Int Int
  deriving (Eq, Ord, Show)
data DeltaState = DeltaState Int DeltaBinds
  deriving (Eq, Show)
type DeltaBinds = [(DeltaId, Delta)]
type M = State DeltaState
eval4 :: Expr → M.Map Name (Double, Delta) → M (Double, Delta)
prop_eval4     = runState (eval4 (Mul (Lit 2) (Var "x"))
                                 (M.singleton "x" (3, DVar (Name "dx"))))
                          (DeltaState 0 [ ])
                 == ((6, DVar (Int 0)),
                     DeltaState 1 [(Int 0, DAdd (DScale 2 (DVar (Name "dx")))
                                                (DScale 3 Zero))])
prop_eval4_Let = runState (eval4 (Let "y" (Mul (Lit 2) (Var "x"))
                                     (Add (Var "y") (Var "y")))
                                 (M.singleton "x" (3, DVar (Name "dx"))))
                          (DeltaState 0 [ ])
                 == ((12, DVar (Int 1)),
                     DeltaState 2 [(Int 1, DAdd (DVar (Int 0))
                                                (DVar (Int 0)))
                                  , (Int 0, DAdd (DScale 2 (DVar (Name "dx")))
                                                (DScale 3 Zero))])

runDelta :: M (Double, Delta) → (Double, Delta)
prop_runDelta = runDelta (eval4 (Let "y" (Mul (Lit 2) (Var "x"))
                                    (Add (Var "y") (Var "y")))
                                (M.singleton "x" (3, DVar (Name "dx"))))
                == (12, DLet (Int 0) (DAdd (DScale 2 (DVar (Name "dx")))
                                           (DScale 3 Zero))
                        (DLet (Int 1) (DAdd (DVar (Int 0))
                                            (DVar (Int 0)))
                              (DVar (Int 1))))
runDelta m = (res, foldl wrap delta binds)
  where ((res, delta), DeltaState _ binds) = runState m (DeltaState 0 [ ])
        wrap body (id, rhs)                = DLet id rhs body
```

```
type DeltaMap = M.Map DeltaId Double
evalDelta :: Double → Delta → DeltaMap → DeltaMap
prop_evalDelta  = evalDelta 100
                            (DLet (Int 0) (DAdd (DScale 2 (DVar (Name "dx")))
                                                (DScale 3 Zero))
                                  (DLet (Int 1) (DAdd (DVar (Int 0))
                                                      (DVar (Int 0)))
                                        (DVar (Int 1))))
                            (M.fromList [(Name "dx", 1), (Name "dz", 42)])
                  == M.fromList [(Name "dx", 401), (Name "dz", 42)]
prop_evalDelta2 = evalDelta 100
                            (DLet (Int 1) (DAdd (DVar (Int 0))
                                                (DVar (Int 0)))
                                  (DVar (Int 1)))
                            (M.fromList [(Name "dx", 1), (Name "dz", 42)])
                  == M.fromList [(Name "dx", 1), (Name "dz", 42), (Int 0, 200)]
prop_evalDelta1 = evalDelta 200
                            (DAdd  (DScale 2 (DVar (Name "dx")))
                                   (DScale 3 Zero))
                            (M.fromList [(Name "dx", 1), (Name "dz", 42)])
                  == M.fromList [(Name "dx", 401), (Name "dz", 42)]
evalDelta _ Zero             um = um
evalDelta x (DAdd u1 u2)     um = evalDelta x u2 (evalDelta x u1 um)
evalDelta x (DScale y u)     um = evalDelta (x × y) u um
evalDelta x (DVar uid)       um = M.insertWith (+) uid x um
evalDelta x (DLet uid u1 u2) um = let um2 = evalDelta x u2 um in
                                  case M.lookup uid um2 of
                                    Nothing → um2
                                    Just x  → evalDelta x u1 (M.delete uid um2)

stepParams :: Params → Params
stepParams params =
  let dualize n (Inertia v _) = (v, DVar (Name n))
      (_, u) = runDelta (eval4 loss (M.mapWithKey dualize params))
      grad = M.mapKeysMonotonic (λ(Name n) → n) (evalDelta 1 u M.empty)
      stepParam u (Inertia v oldMomentum) =
        let newMomentum = 0.9 × oldMomentum − 0.1 × u
        in Inertia (v + newMomentum) newMomentum
  in M.union (M.intersectionWith stepParam grad params)
             (M.map (stepParam 0) (M.difference params grad))
```

$$
\begin{array}{l}
\mathit{deltaLet} :: \mathit{Delta} \to M\ \mathit{Delta} \\
\mathit{deltaLet}\ \mathit{delta} = \mathit{state}\ (\lambda(\mathit{DeltaState}\ \mathit{next}\ \mathit{deltas}) \to \\
\qquad\qquad\qquad (\mathit{DVar}\ (\mathit{Int}\ \mathit{next}), \mathit{DeltaState}\ (\mathit{next}+1)\ ((\mathit{Int}\ \mathit{next}, \mathit{delta}) : \mathit{deltas})))
\end{array}
$$

$$
\begin{array}{lll}
\mathit{eval}_4\ (\mathit{Lit}\ v) & \_ & = \mathit{return}\ (v, \mathit{Zero}) \\
\mathit{eval}_4\ (\mathit{Add}\ e_1\ e_2) & \mathit{env} & = \mathbf{do}\ (v_1, u_1) \leftarrow \mathit{eval}_4\ e_1\ \mathit{env} \\
 & & \qquad (v_2, u_2) \leftarrow \mathit{eval}_4\ e_2\ \mathit{env} \\
 & & \qquad u \leftarrow \mathit{deltaLet}\ (\mathit{DAdd}\ u_1\ u_2) \\
 & & \qquad \mathit{return}\ (v_1 + v_2, u) \\
\mathit{eval}_4\ (\mathit{Mul}\ e_1\ e_2) & \mathit{env} & = \mathbf{do}\ (v_1, u_1) \leftarrow \mathit{eval}_4\ e_1\ \mathit{env} \\
 & & \qquad (v_2, u_2) \leftarrow \mathit{eval}_4\ e_2\ \mathit{env} \\
 & & \qquad u \leftarrow \mathit{deltaLet}\ (\mathit{DAdd}\ (\mathit{DScale}\ v_1\ u_2)\ (\mathit{DScale}\ v_2\ u_1)) \\
 & & \qquad \mathit{return}\ (v_1 \times v_2, u) \\
\mathit{eval}_4\ (\mathit{Var}\ n) & \mathit{env} & = \mathbf{do}\ \mathit{return}\ (\mathit{env}\ M.!\ n) \\
\mathit{eval}_4\ (\mathit{Let}\ n\ \mathit{rhs}\ e) & \mathit{env} & = \mathbf{do}\ v \leftarrow \mathit{eval}_4\ \mathit{rhs}\ \mathit{env} \\
 & & \qquad \mathit{eval}_4\ e\ (M.\mathit{insert}\ n\ v\ \mathit{env})
\end{array}
$$

DeltaBinds can be a mutable array that grows in *deltaLet*.
DeltaMap can be a mutable array that shrinks in *evalDelta*.

REFERENCES

Claessen, Koen, 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*, vol. 35(9) of *ACM SIGPLAN Notices*, 268–279. New York: ACM Press.

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.