Fortgeschrittene Funktionale Programmierung

10. und 11. Vorlesung

Janis Voigtländer

Universität Bonn

Wintersemester 2015/16

Nochmal Beispiele zu „puren“ Monaden

Weiter (aber Datentyp etwas abgespeckt):

```
data Expr :: * → * where
  Lit  :: Int → Expr Int
  Var  :: String → Expr Int
  Add :: Expr Int → Expr Int → Expr Int
  Sub :: Expr Int → Expr Int → Expr Int
```

Angenommen, wir wollen zur Normalisierung eines Ausdrucks darin vorkommende Variablen durchnummerieren. Also, zum Beispiel:

```
   Add (Sub (Var "a") (Lit 3)) (Var "b")
⇝ Add (Sub (Var "x1") (Lit 3)) (Var "x2")
```

Nochmal Beispiele zu „puren“ Monaden

Allgemeines Verwalten/Propagieren eines Zustands:

$$\textbf{newtype}\ \mathrm{State}\ s\ a = \mathrm{State}\ \{\mathtt{runState} :: s \to (a, s)\}$$

$$\begin{aligned}
&\textbf{instance}\ \mathrm{Monad}\ (\mathrm{State}\ s)\ \textbf{where}\\
&\quad \mathtt{return}\ a \quad\ = \mathrm{State}\ (\lambda s \to (a, s))\\
&\quad \mathrm{State}\ k \gg\!= f = \mathrm{State}\ (\lambda s \to \textbf{let}\ (a, s') = k\ s\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{in}\ \mathtt{runState}\ (f\ a)\ s')
\end{aligned}$$

$$\begin{aligned}
&\mathtt{get} :: \mathrm{State}\ s\ s\\
&\mathtt{get} = \mathrm{State}\ (\lambda s \to (s, s))
\end{aligned}$$

$$\begin{aligned}
&\mathtt{put} :: s \to \mathrm{State}\ s\ ()\\
&\mathtt{put}\ s = \mathrm{State}\ (\lambda_ \to ((), s))
\end{aligned}$$

$$\begin{aligned}
&\mathtt{evalState} :: \mathrm{State}\ s\ a \to s \to a\\
&\mathtt{evalState}\ m\ s = \mathtt{fst}\ (\mathtt{runState}\ m\ s)
\end{aligned}$$

(Dann zum Beispiel: StrangeEnv $\approx$ State (String $\to$ Int).)

Nochmal Beispiele zu „puren“ Monaden

Jetzt:

```
rename :: Expr t → State Int (Expr t)
rename (Lit n)  = return (Lit n)
rename (Var _) = do i ← get
                    put (i + 1)
                    return (Var ('x' : show i))
rename (Add e1 e2) = liftM2 Add (rename e1) (rename e2)
rename (Sub e1 e2) = liftM2 Sub (rename e1) (rename e2)
```

Tut richtig:

```
> evalState (rename (Add (Sub (Var "a") (Lit 3)) (Var "b"))) 1
Add (Sub (Var "x1") (Lit 3)) (Var "x2")
```

Aber, tut „falsch“:

```
> evalState (rename (Add (Sub (Var "a") (Lit 3)) (Var "a"))) 1
Add (Sub (Var "x1") (Lit 3)) (Var "x2")
```

Nochmal Beispiele zu „puren“ Monaden

Mit präziserem Bookkeeping:

```
rename :: Expr t → State [String] (Expr t)
rename (Lit n)  = return (Lit n)
rename (Var v) = do names ← get
                    if elem v names
                      then let Just i = lookup v (zip names [1..])
                           in return (Var ('x' : show i))
                      else do let i = length names + 1
                              put (names ++ [v])
                              return (Var ('x' : show i))
rename (Add e1 e2) = liftM2 Add (rename e1) (rename e2)
rename (Sub e1 e2) = liftM2 Sub (rename e1) (rename e2)
```

```
> evalState (rename (Add (Sub (Var "a") (Lit 3)) (Var "b"))) []
Add (Sub (Var "x1") (Lit 3)) (Var "x2")
> evalState (rename (Add (Sub (Var "a") (Lit 3)) (Var "a"))) []
Add (Sub (Var "x1") (Lit 3)) (Var "x1")
```

Ein Beispiel zur „Kombination von Effekten“

Angenommen, wir wollen verschiedene Normalisierungen zur Auswahl haben, insbesondere keine willkürliche Festlegung auf Nummerierung „von links nach rechts“ bei Add und Sub?

```
newtype StateNondet s a = StateNondet {runSND :: s -> [(a, s)]}

instance Monad (StateNondet s) where
  return a = StateNondet (λs -> [(a, s)])
  StateNondet k >>= f
    = StateNondet (λs -> concatMap (λ(a, s') -> runSND (f a) s')
                                   (k s))

get :: StateNondet s s
get = StateNondet (λs -> [(s, s)])

put :: s -> StateNondet s ()
put s = StateNondet (λ_ -> [((), s)])

(|||) :: StateNondet s a -> StateNondet s a -> StateNondet s a
StateNondet k1 ||| StateNondet k2 = StateNondet (λs -> k1 s ++ k2 s)
```

Ein Beispiel zur „Kombination von Effekten“

```
rename :: Expr t → StateNondet [String] (Expr t)
rename (Lit n)   = return (Lit n)
rename (Var v) = ...
rename (Add e1 e2) = do e1' ← rename e1
                        e2' ← rename e2
                        return (Add e1' e2')
                     |||
                     do e2' ← rename e2
                        e1' ← rename e1
                        return (Add e1' e2')
rename (Sub e1 e2) = ...

> runSND (rename (Add (Var "a") (Var "b"))) []
[(Add (Var "x1") (Var "x2"), ["a", "b"]),
 (Add (Var "x2") (Var "x1"), ["b", "a"])]
```

(Übrigens: LParser $\approx$ StateNondet String.)

Da war doch noch was...

Aus der „Alpen“-Aufgabe des Bundeswettbewerbs:

3. Entwirf und implementiere einen Algorithmus, der für gegebenes N eine später zu spezifizierende Prozedur P nacheinander mit jedem Gebirgszug der Länge N als Argument aufruft.

   a) Benutze deinen Algorithmus mit einem geeigneten P, um alle Gebirgszüge der Länge 6 auszugeben, und zeige deine Ausgabe.

   b) Benutze deinen Algorithmus mit einem anderen P, um die Anzahl der Gebirgszüge der Länge 16 zu bestimmen.

```
perform :: Monad m => Int -> (Alpen -> m ()) -> m ()
perform n p = mapM_ p (generate n)
  where mapM_ :: Monad m => (a -> m b) -> [a] -> m ()
        mapM_ f []     = return ()
        mapM_ f (x:xs) = do f x
                            mapM_ f xs
```

Dann, 3.b), Bestimmung Anzahl der Gebirgszüge:

```
count :: Int → Int
count n = execState (perform n (λ_ → tick)) 0
  where tick :: State Int ()
        tick = get >>= λn → put (n + 1)
        execState :: State s a → s → s
        execState m s = snd (runState m s)
```

```
> count 16
853467
```

(„Allerdings“: keine echte veränderliche Variable im Sinne imperativer Programmierung benutzt.)

Für 3.a), anschauliche Ausgabe der Gebirgszüge:

```
output :: Int → IO ()
output n = perform n (putStrLn ∘ show)
  where show :: Alpen → String
        show = ...
```

Also, ... IO.

Beobachtungen, schon aus DP:

- Integration von Ein-/Ausgabe in eine rein funktionale Sprache mit lazy Auswertung ist notwendigerweise nicht trivial.
- In Haskell wurde dieses Problem (nach einigen Um-/Abwegen) darüber gelöst, diese Effekte in einer Monade zu kapseln.
- Anders als die bisherigen Beispiele lässt sich die IO-Monade nicht selbst innerhalb Haskell implementieren.

Verwendung von IO

Das Monad-Interface stellt exakt die richtigen Operationen zur Verfügung, die man zur sinnvollen Sequentialisierung braucht.

Für die eigentlichen Effekte gibt es eine Vielzahl von Primitiven:

- `getChar`, `putChar`, `putStrLn`, `openFile`, ...
- `getArgs`, `getEnvironment`, `getClockTime`, ...
- `throwIO`, `catch`, ...
- `forkIO`, ...
- `newIORef`, `readIORef`, `writeIORef`, ...
- `randomIO`, ...
- GUI frameworks

Es gibt keine (sichere) Möglichkeit, aus einem Wert vom Typ „IO *a*“ das (bzw. ein) „*a*“ zu extrahieren.

Interessantes Feature: (imperative) Referenzen

Zur Erinnerung:

```
perform :: Monad m ⇒ Int → (Alpen → m ()) → m ()
perform n p = mapM_ p (generate n)

count :: Int → Int
count n = execState (perform n (λ_ → tick)) 0
  where tick :: State Int ()
        tick = get ≫= λn → put (n + 1)
```

(„Allerdings“: keine echte veränderliche Variable im Sinne imperativer Programmierung benutzt.)

Mit echten Referenzen (**import** Data.IORef):

```
count :: Int → IO Int
count n = do ref ← newIORef 0
             perform n (λ_ → tick ref)
             readIORef ref
  where tick ref = readIORef ref ≫= λn → writeIORef ref (n + 1)
```

Interessantes Feature: (imperative) Referenzen

Mit echten Referenzen (**import** Data.IORef):

```
count :: Int → IO Int
count n = do ref ← newIORef 0
             perform n (λ_ → tick ref)
             readIORef ref
  where tick ref = readIORef ref >>= λn → writeIORef ref (n + 1)
```

Oder auch:

```
count :: Int → IO Int
count n = do ref ← newIORef 0
             let get = readIORef ref
             let put = writeIORef ref
             let tick = get >>= λn → put (n + 1)
             perform n (λ_ → tick)
             get
```

Interessantes Feature: (imperative) Referenzen

Mit echten Referenzen (**import** Data.IORef):

```
count :: Int → IO Int
count n = do ref ← newIORef 0
             perform n (λ_ → tick ref)
             readIORef ref
  where tick ref = readIORef ref >>= λn → writeIORef ref (n + 1)
```

```
newIORef    :: a → IO (IORef a)
readIORef   :: IORef a → IO a
writeIORef  :: IORef a → a → IO ()
```

```
tick :: IORef Int → IO ()
```

Umgang mit mehreren Referenzen

```
swap :: IORef a → IORef a → IO ()
swap v w = do a ← readIORef v
              b ← readIORef w
              writeIORef v b
              writeIORef w a

f :: Int → IO Int
f n = do v ← newIORef 0
         w ← newIORef 0
         let go n = when (n > 0) (do tick v
                                     swap v w
                                     go (n − 1))
         go n
         readIORef v
```

Umgang mit mehreren Referenzen

```
f :: Int → IO Int
f n = do v ← newIORef 0
         w ← newIORef 0
         let go n = when (n > 0) (do tick v
                                     swap v w
                                     go (n − 1))
         go n
         readIORef v
```

Man beachte:

- Die beiden Vorkommen von `newIORef 0` führen verschiedene Referenzen ein! Diskussion:

```
do v ← newIORef 0          do let new = newIORef 0          do new ← newIORef 0
   w ← newIORef 0             v ← new                          let v = new
   ...              vs.       w ← new              vs.         let w = new
                              ...                              ...
```

- Man kommt nicht so ohne Weiteres von `f :: Int → IO Int` zu `f :: Int → Int` (oder von `count :: Int → IO Int` zu `Int → Int`).

Zustandskapselung, ST statt IO

```
import Control.Monad.ST
import Data.STRef.Strict

tick ref = readSTRef ref >>= λn → writeSTRef ref (n + 1)

swap v w = do a ← readSTRef v
              b ← readSTRef w
              writeSTRef v b
              writeSTRef w a

f :: Int → Int
f n = runST (do v ← newSTRef 0
                w ← newSTRef 0
                let go n = when (n > 0) (do tick v
                                            swap v w
                                            go (n − 1))
                go n
                readSTRef v)
```

ST-Zustandskapselung, ein Blick auf die Typen

```
newSTRef   :: a → ST t (STRef t a)
readSTRef  :: STRef t a → ST t a
writeSTRef :: STRef t a → a → ST t ()
runST      :: (∀t.ST t a) → a
```

Vergleiche:

```
newIORef   :: a → IO (IORef a)
readIORef  :: IORef a → IO a
writeIORef :: IORef a → a → IO ()
```

Der zusätzliche Typparameter t bei ST dient dem eindeutigen „threading“/Zuordnung von Referenzen zu ihrem Gültigkeitsbereich.

ST-Zustandskapselung, ein Blick auf die Typen

```
newSTRef    :: a → ST t (STRef t a)
readSTRef   :: STRef t a → ST t a
writeSTRef  :: STRef t a → a → ST t ()
runST       :: (∀t.ST t a) → a
```

Mit obigen Typen also:

```
tick :: STRef t Int → ST t ()
swap :: STRef t a → STRef t a → ST t ()
```

Nun zum Beispiel auch möglich:

```
count :: Int → Int
count n = runST (do ref ← newSTRef 0
                    perform n (λ_ → tick ref)
                    readSTRef ref)
```

Challenge: Interpreter mit echten Referenzen

Ausgehend von:

```
data Expr :: * → * where
  Lit   :: Int → Expr Int
  Var   :: String → Expr Int
  Add :: Expr Int → Expr Int → Expr Int
  ...

type StrangeEnv a = (String → Int) → (a, String → Int)

instance Monad StrangeEnv where
  return a = λenv → (a, env)
  m >>= f  = λenv → let (a, env') = m env
                    in f a env'

eval :: Expr t → StrangeEnv t
eval (Lit n)      = return n
eval (Var s)      = λenv → (env s, λs' → if s == s' then 0 else env s')
eval (Add e1 e2) = liftM2 (+) (eval e1) (eval e2)
...
```