fnord.hs

Sitzung 3: IO & Monaden

3. September 2017

Problem

```
-- Stell dir vor, es gaebe:
readLine :: String
-- und:
printLine :: String → ()

-- Was macht:
foo = print (readLine ++ readLine)

-- Was macht:
fuu = print (a ++ a)
  where a = readLine
```

Funktionale Reinheit

- Referenzielle Transparenz: Ausdrücke lassen sich mit let oder where herausfaktorisieren ohne die Semantik zu ändern
- Freiheit von Seiteneffekten: Funktion liefert immer dasselbe Ergebnis, wenn sie das selbe Argument bekommt; weiterhin spielt es keine Rolle, wie häufig eine Funktion ausgewertet wird

Stream-based IO

```
data Request = AppendChannel Name String
             | ReadChannel Name
             | AppendFile Name String
             | WriteFile Name String
             | ReadFile Name
             | DeleteFile Name

data Response = Success
              | Return String
              | Failure String

type Behaviour = [Response] → [Request]
```

Stream-based IO

```
main :: Behaviour
main responses =
  AppendChannel "stdout"
    "Geben␣Sie␣bitte␣eine␣Zahl␣ein:␣" :
  ReadChannel "stdin" :
  case responses of
    (Success : Return zahlstr : more) →
      let zahl = read zahlstr
      in AppendChannel "stdout" (
           "Das␣Doppelte␣Ihrer␣Zahl␣ist:␣"
           ++ show (zahl*2)) : []
    (Failure fehler : _ : more) →
      AppendChannel "stderr" fehler : []
    (_ : Failure fehler : more) →
      AppendChannel "stderr" fehler : []
```

Continuation-based IO

```
type FailCont = String → Behaviour
type RetCont = String → Behaviour
type SuccCont = Behaviour

appendChannel :: Name → String →
    FailCont → SuccCont → Behaviour
appendChannel chan cts fail succ resps =
     AppendChannel chan cts :
     case resps of
        Success:more → succ more
        Failure err:more → fail err more

putStr = appendChannel "stdout"
putStrLn str = putStr (str++"\n")
```

Continuation-based IO

```
readChannel :: Name →
      FailCont → RetCont → Behaviour
readChannel chan fail ret resps =
    ReadChannel chan :
    case resps of
      Return str:more → ret str more
      Failure err:more → fail err more

getLine = readChannel "stdin"
```

Continuation-based IO

```
exit :: Behaviour
exit = const []

die :: FailCont
die msg =
    putStrLn ("Error: "++msg) (const exit) exit

main :: Behaviour
main =
  putStr "Geben Sie bitte eine Zahl ein: " die $
  getLine die $ \zahlstr →
  putStrLn ("Das Doppelte Ihrer Zahl ist: "
          ++ show (read zahlstr*2)) die exit
```

Monadisches IO

```
newtype CIO a = CIO { runCIO ::
        (a → Behaviour) → Behaviour }

return :: a → CIO a
return a = CIO $ \f → f a

(>>=) :: CIO a → (a → CIO b) → CIO b
CIO m >>= f = CIO $ \bf → m (\a → runCIO (f a) bf)

(>>) :: CIO a → CIO b → CIO b
a >> b = a >>= const b
```

Monadisches IO

```
putStr' :: String → CIO ()
putStr' str = CIO $ \f → putStr str die (f ())
putStrLn' str = putStr' (str++"\n")

getLine' :: CIO String
getLine' = CIO $ getLine die

main :: CIO ()
main =
  putStr' "Geben Sie bitte eine Zahl ein: " >>
  getLine' >>= \zahlstr →
  putStrLn' ("Das Doppelte Ihrer Zahl ist: "
             ++ show (read zahlstr*2))
```

Monadisches IO

```
getRandom :: CIO Int
getRandom = return 4 -- chosen by fair dice roll

main :: CIO ()
main =
  putStr' "Geben Sie bitte eine Zahl ein: " >>
  getLine' >>= \zahlstr →
  getRandom >>= \r →
  putStrLn' ("Das 4-fache Ihrer Zahl ist: "
             ++ show (read zahlstr*r))
```

Monaden

```
class Applicative m ⇒ Monad m where
  return :: a → m a
  (>>=) :: m a → (a → m b) → m b

instance Monad CIO where ...

instance Monad Maybe where
  return = Just
  Nothing >>= _ = Nothing
  Just a >>= f = f a
```

Monaden

```
Prelude> Just 5 >>= \a → Just (a*5)
Just 25
Prelude> Nothing >>= \a → Just (a*5)
Nothing
```

Monaden

```
concatMap f xs = concat $ map f xs

instance Monad [] where
  return = (:[])
  (>>=) = concatMap

instance Monad ((->) a) where
  return = const
  f >>= k = \r -> k (f r) r

instance Monoid a => Monad ((,) a) where
  return a = (mempty, a)
  (u, a) >>= k =
  case k a of
    (v, b) -> (u `mappend` v, b)
```

Monaden

```
Prelude> let tell s = (s,())
Prelude> tell "fnord" >> tell "eingang" >> return 5
("fnordeingang",5)
Prelude> let ask = id
Prelude> (ask >>= \a → return (a*2)) 2
4
Prelude> [1,2,3] >>= \a → [a,a*2]
[1,2,2,4,3,6]
```

do-Notation

- **do** { a $\leftarrow$ m; n } desugart zu m >>= \a $\rightarrow$n
- **do** { m; n } desugart zu m >> n

do-Notation

```
Prelude> do { tell "fnord"; tell "eingang"; return 5 }
("fnordeingang",5)
Prelude> (do { a ← ask; return (a*2) }) 2
4
Prelude> do { a ← [1,2,3]; [a,a*2] }
[1,2,2,4,3,6]
```

Monadisches IO

```
main :: IO ()
main = do
  putStr "Geben Sie bitte eine Zahl ein: "
  zahlstr ← getLine
  putStrLn' ("Das Doppelte Ihrer Zahl ist: "
             ++ show (read zahlstr*2))
```