Haskell Workshop - Free Monads

Carsten König

Juni 2021

Freie Monaden

Wollen kleine Schnittstelle um Programme schreiben zu können, die Texte einlesen und ausgeben (z.B. über die Console).

Dabei soll der Ablauf des Programms über einen Datentyp abstrahiert werden.

1. Schritt

```haskell
data Teletyper
  = Write String Teletyper
  | Read (String -> Teletyper)
  | Done

sayHello :: Teletyper
sayHello =
  Write "What is your name?" (Read (\name -> Write ("Hello " ++ name) Done))

run :: Teletyper -> IO ()
run Done =
  pure ()
run (Write text cont) = do
  putStrLn text
  run cont
run (Read cont) = do
  s <- getLine
  run (cont s)
```

2. Schritt

Um das Programm (`sayHello`) zu definieren soll die Monad-`do` Umgebung genutzt werden können.

`Teletyper` muss also zur Monade werden (die Aktionen aneinanderreiht). Dazu muss der Typ um einen generischen Rückgabetyp erweitert werden und entsprechende Instanzen erstellt werden.

```haskell
{-# LANGUAGE DeriveFunctor #-}
module Teletyper where

import Control.Monad ((>=>), ap)

data Teletyper a
  = Write String (Teletyper a)
  | Read (String -> Teletyper a)
  | Done a
  deriving Functor

instance Applicative Teletyper where
  pure = Done
  (<*>) = ap

instance Monad Teletyper where
  (Done x) >>= f = f x
  (Write txt cont) >>= f = Write txt (cont >>= f)
  (Read cont) >>= f = Read (cont >=> f)

-- vereinfachende Funktionen

writeTR :: String -> Teletyper ()
writeTR txt = Write txt (Done ())

readTR :: Teletyper String
readTR = Read Done

sayHello :: Teletyper ()
sayHello = do
  writeTR "What is your name?"
  name <- readTR
  writeTR ("Hello " ++ name)

run :: Teletyper a -> IO a
run (Done a) =
  pure a
run (Write text cont) = do
  putStrLn text
  run cont
run (Read cont) = do
  s <- getLine
  run (cont s)
```

3. Schritt

Trenne die Monade in einen Functor für das Verhalten und eine Monade, die Rekursion/Rückgabe übernimmt:

```haskell
{-# LANGUAGE DeriveFunctor #-}
module Teletyper.Version3 where

import Control.Monad ((>=>), ap)

data TeletyperF r
  = Write String r
  | Read (String -> r)
  deriving Functor

data Teletyper a
  = Done a
  | Rec (TeletyperF (Teletyper a))
  deriving Functor

instance Applicative Teletyper where
  pure = Done
  (<*>) = ap

instance Monad Teletyper where
  (Done x) >>= f = f x
  (Rec ttf) >>= f = Rec (fmap (>>= f) ttf)

writeTR :: String -> Teletyper ()
writeTR txt = Rec (Write txt (Done ()))
```

```haskell
readTR :: Teletyper String
readTR = Rec (Read Done)

sayHello :: Teletyper ()
sayHello = do
  writeTR "What is your name?"
  name <- readTR
  writeTR ("Hello " ++ name)

run :: Teletyper a -> IO a
run (Done a) =
  pure a
run (Rec (Write text cont)) = do
  putStrLn text
  run cont
run (Rec (Read cont)) = do
  s <- getLine
  run (cont s)
```

4. Schritt

Die Interpretation (manchmal *Algebra* genannt) der Funktionalität auslagern.

```haskell
{-# LANGUAGE DeriveFunctor #-}
module Teletyper.Version4 where

import Control.Monad ((>=>), ap)

data TeletyperF r
  = Write String r
  | Read (String -> r)
  deriving Functor

runF :: TeletyperF (IO a) -> IO a
runF (Write text cont) = putStrLn text >> cont
runF (Read cont) = getLine >>= cont

data Teletyper a
  = Done a
  | Rec (TeletyperF (Teletyper a))
  deriving Functor


instance Applicative Teletyper where
  pure = Done
  (<*>) = ap

instance Monad Teletyper where
  (Done x) >>= f = f x
  (Rec (Write txt cont)) >>= f = Rec (Write txt (cont >>= f))
  (Rec (Read cont)) >>= f = Rec (Read (\s -> cont s >>= f))

writeTR :: String -> Teletyper ()
writeTR txt = Rec (Write txt (Done ()))

readTR :: Teletyper String
readTR = Rec (Read Done)

sayHello :: Teletyper ()
sayHello = do
  writeTR "What is your name?"
  name <- readTR
  writeTR ("Hello " ++ name)

run :: Teletyper a -> IO a
run (Done a) = pure a
run (Rec tf) = runF (fmap run tf)
```

Freie Monade

Wenn man genau hinsieht merkt man, dass die Funktor-Eigenschaft ausreicht um die Monade zu definieren. Wir können das abstrahieren:

```
{-# LANGUAGE DeriveFunctor #-}
module Teletyper.Version5 where

import Control.Monad ((>=>), ap)

data TeletyperF r
  = Write String r
  | Read (String -> r)
  deriving Functor

runF :: TeletyperF (IO a) -> IO a
runF (Write text cont) = putStrLn text >> cont
runF (Read cont) = getLine >>= cont


data Free f a
  = Pure a
  | Free (f (Free f a))
  deriving Functor

instance Functor f => Applicative (Free f) where
  pure = Pure
  (<*>) = ap

instance Functor f => Monad (Free f) where
  (Pure x) >>= f = f x
  (Free m) >>= f = Free (fmap (>>= f) m)

runFree :: (Monad m, Functor f) => (f (m a) -> m a) -> Free f a -> m a
runFree _ (Pure a) = pure a
runFree runFunctor (Free f) = runFunctor (fmap (runFree runFunctor) f)

type Teletyper = Free TeletyperF

writeTR :: String -> Teletyper ()
writeTR txt = Free (Write txt (Pure ()))

readTR :: Teletyper String
readTR = Free (Read Pure)

sayHello :: Teletyper ()
sayHello = do
  writeTR "What is your name?"
  name <- readTR
  writeTR ("Hello " ++ name)
```

```haskell
run :: Teletyper a -> IO a
run = runFree runF
```

Transformer

Alles was wir hier gemacht haben findet man z.B. im `free`-Package

Dort kann man sich auch einen entsprechenden Transformer ansehen.

Einige Effektsysteme beruhen auf dieser Technik - für den Produktiven Ansatz sind die beliebtesten:

- Polysemy
- eff - persönlich denke ich das wird sich durchsetzen, Performance durch Änderungen am GHC
- fused-effects

Hier das Beispiel von oben für `fused-effects`:

```haskell
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE GADTs #-}
{-# LANGUAGE GeneralizedNewtypeDeriving #-}
{-# LANGUAGE KindSignatures #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE TypeOperators #-}
{-# LANGUAGE UndecidableInstances #-}
module Teletyper.VersionFused where

-- siehe: https://github.com/fused-effects/fused-effects/blob/master/docs/defining_effects.md

import Control.Algebra (Algebra, Has, alg, send, (:+:)(..))
import Control.Monad.IO.Class (MonadIO(liftIO))
import Data.Kind

-- Syntax / Effect
data Teletype (m :: Type -> Type) k where
  Read  ::           Teletype m String
  Write :: String -> Teletype m ()

readTR :: Has Teletype sig m => m String
readTR = send Read

writeTR :: Has Teletype sig m => String -> m ()
writeTR s = send (Write s)

-- Carrier (Semantic) Effect
newtype TeletypeIOC m a = TeletypeIOC { runTeletypeIO :: m a }
  deriving (Applicative, Functor, Monad, MonadIO)

-- Algebra-Instanz für den Carrier (Interpretation)
instance (MonadIO m, Algebra sig m) => Algebra (Teletype :+: sig) (TeletypeIOC m) where
  alg hdl sig ctx = case sig of
    L Read      -> (<$ ctx) <$> liftIO getLine
    L (Write s) -> ctx      <$  liftIO (putStrLn s)
    R other     -> TeletypeIOC (alg (runTeletypeIO . hdl) other ctx)

-- Sag Hallo

sayHello :: Has Teletype sig m => m ()
sayHello = do
  writeTR "What is your name?"
  name <- readTR
  writeTR ("Hello " ++ name)
```

```
run :: TeletypeIOC m a -> m a
run = runTeletypeIO
```