Haskell Workshop - Monaden-Transformer

Carsten König

Juni 2021

Standardmonaden

Either

wird gerne für Fehler benutzt (ROP)

- `Left ...` zeigt Fehler an
- `Right ...` zeigt Ergebnis an

```haskell
newtype Except e a = Except { runExcept :: Either e a }
  deriving (Functor, Applicative)


instance Monad (Except e) where
  Except (Left err) >>= _ = Except (Left err)
  Except (Right  a) >>= f = f a


throwError :: e -> Except e a
throwError err = Except (Left err)


catchError :: Except e a -> (e -> Except e a) -> Except e a
catchError m handler =
  case runExcept m of
    Left err -> handler err
    Right  a -> return a


exceptExample :: Int -> Except String Int
exceptExample 0 = return 0
exceptExample 2 = return 1
exceptExample n
  | even n = do
    halfRes <- exceptExample (n `div` 2)
    return $ 1 + halfRes
  | otherwise  = throwError "keine Potenz von 2"


-- catching:
runExcept $ exceptExample 33 `catchError` (const $ return (-1))
```

Reader

wird gerne verwendet um Konfiguration/Umgebung zur Verfügung zu stellen

```haskell
newtype Reader r a = Reader { runReader :: r -> a }
  deriving (Functor, Applicative)
```

```haskell
instance Monad (Reader r) where
  m >>= f =
    Reader $ \r -> runReader (f (runReader m r)) r


ask :: Reader r r
ask = Reader id


reader :: (r -> a) -> Reader r a
reader = Reader


local :: (r -> r) -> Reader r a -> Reader r a
local adj m = Reader $ \r -> runReader m (adj r)
```

State

```haskell
newtype State s a = State { runState :: s -> (a,s) }
  deriving (Functor)


instance Applicative (State s) where
  pure a = State $ \s -> (a,s)
  sf <*> sx = sf >>= (\f -> f <$> sx)


instance Monad (State s) where
  m >>= f = State $ \s ->
    let (a, s') = runState m s
    in runState (f a) s'


get :: State s s
get = State $ \s -> (s,s)


put :: s -> State s ()
put s = State $ \_ -> ((), s)


modify :: (s -> s) -> State s ()
modify adj = State $ \ s -> ((), adj s)


gets :: (s -> a) -> State s a
gets g = State $ \ s -> (g s, s)


evalState :: State s a -> s -> a
evalState m = fst . runState m


execState :: State s a -> s -> s
execState m = snd . runState m
```

Transformer

Monaden sind leider nicht wie z.B. Funktoren komposierbar. Deshalb *stacken* wir sie mit Transformatoren.

Der Monadentransformator erweitert sozusagen einen Monaden um die Fähigkeiten eines anderen.

EitherT

```
newtype ExceptT e m a = ExceptT { runExceptT :: m (Either e a) }
  deriving (Functor)


instance Monad m => Applicative (ExceptT e m) where
  pure = ExceptT . return . Right
  mf <*> mx = mf >>= (\f -> f <$> mx)


instance Monad m => Monad (ExceptT e m) where
  (ExceptT m) >>= f =
    ExceptT $ m >>= (\case (Left err) -> return (Left err)
                           (Right  a) -> runExceptT (f a))


liftE :: Monad m => m a -> ExceptT e m a
liftE m = ExceptT $ Right <$> m


throwErrorT :: Monad m => e -> ExceptT e m a
throwErrorT err = ExceptT (return $ Left err)


catchErrorT :: Monad m => ExceptT e m a -> (e -> ExceptT e m a) -> ExceptT e m a
catchErrorT m handler = ExceptT $ do
  res <- runExceptT m
  case res of
    Left err -> runExceptT $ handler err
    Right  a -> return (Right a)
```

Klassen

damit nicht überall `lift` benutzt werden muss, führt **MTL** noch Typ-Klassen ein. Zum Beispiel gibt es einen `MonadError`:

MonadError

siehe Control.Monad.MonadError

```
class Monad m => MonadError e m | m -> e where
  throw :: e -> m a
  catch :: m a -> (e -> m a) -> m a


instance Monad m => MonadError e (ExceptT e m) where
  throw = throwErrorT
  catch = catchErrorT
```

Über die *funktionale Abhängigkeit* `| m -> e` wird gesagt, dass der Monade den Typ des Fehlers **bestimmen** muss. Danach werden für alle anderen Klassen in der MTL Instanzen erstellt, die im wesentlichen eine rekursive Auflösung (über `lift`) bis zu einer `ExceptT` (oder `ErrorT`) Instanz vornimmt.

Instanzen findet man hier

MonadReader

siehe Control.Monad.Reader

```
class Monad m => MonadReader r m | m -> r where
  ask :: m r
  local :: (r -> r) -> m a -> m a
  reader :: (r -> a) -> m a

-- Synonym für reader
asks :: MonadReader r m => (r -> a) -> m a
```

MonadState

siehe Control.Monad.State

```
class Monad m => MonadState s m | m -> s where
  get :: m s
  put :: s -> m ()
  state : (s -> (a, s)) -> m a

gets :: MonadState s m => (s -> a) -> m a

modify :: MonadState s m => (s -> s) -> m ()
-- strikte Variante im neuen Zustand
modify' ...
```

MonadTrans

`lift` funktioniert mit allen Transformatoren

siehe Control.Monad.Trans.Class

```
class MonadTrans t where
  lift :: Monad m => m a -> t m a
```

MonadIO

`liftIO` funktioniert mit allen Transformatoren

siehe Control.Monad.IO.Class

```
class Monad m => MonadIO m where
  liftIO :: IO a -> m a
```

Beispiel

```
import Control.Monad.State
import Control.Monad.Except

import Data.Map.Strict as Map

class Monad m => EventStoreMonad ev m | m -> ev where
  newAggregate :: m Int
  addEvent :: Int -> ev -> m ()
  getEvents :: Int -> m [ev]
```

```haskell
newtype MemoryStoreMonad ev a = MemoryStoreMonad (ExceptT String (State (Map.Map Int [ev])) a)
  deriving (Functor, Applicative, Monad, MonadState (Map.Map Int [ev]), MonadError String)


runMemory :: Map.Map Int [ev] -> MemoryStoreMonad ev a -> (Either String a, Map Int [ev])
runMemory map (MemoryStoreMonad m )=
  flip runState map $ runExceptT m


instance EventStoreMonad ev (MemoryStoreMonad ev) where
  newAggregate = do
    key <- (+1) <$> gets Map.size
    modify (Map.insert key [])
    return key

  getEvents key = do
    lookupRes <- gets (Map.lookup key)
    case lookupRes of
      Nothing -> throwError ("Key " ++ show key ++ " nicht im Store")
      Just evs -> return evs

  addEvent key ev =
    modify (Map.insertWith (++) key [ev])
```

interessante Links

- Mark P. Jones: Functional Programming with Overloading and Higher-Order Polymorphism