Monad Transformers

Monaden kombinieren

Christian Höner zu Siederdissen
christian.hoener.zu.siederdissen@uni-jena.de

Theoretische Bioinformatik, Bioinformatik Uni Jena

Dezember 2023

Identity

```haskell
newtype Identity a = Identity { runIdentity :: a }

instance Functor Identity where
  fmap :: (a->b) -> Identity a -> Identity b
  fmap f = Identity . f . runIdentity

instance Applicative Identity where
  pure :: a -> Identity a
  pure = Identity
  (<*>) :: Identity (a->b) -> Identity a -> Identity b
  Identity f <*> Identity a = Identity (f a)

instance Monad Identity where
  return = pure
  (>>=) :: Identity a -> (a -> Identity b) -> Identity b
  Identity a >>= amb = amb a
```

StateT

```haskell
newtype StateT s m a = StateT {runStateT :: s -> m (a,s)}

instance Functor m => Functor (StateT s m) where
  fmap f m = StateT $ \s ->
    fmap (\ (a,t) -> (f a, t)) $ runStateT m s

instance (Functor m, Monad m) => Applicative (StateT s m) where
  pure a = StateT $ \s -> return (a,s)
  StateT mf <*> StateT mx = StateT $ \s -> do
    (f,t) <- mf s
    (x,u) <- mx t
    return (f x, u)
```

StateT

```haskell
newtype StateT s m a = StateT {runStateT :: s -> m (a,s)}

instance Functor m => Functor (StateT s m) where
  fmap f m = StateT $ \s ->
    fmap (\ (a,t) -> (f a, t)) $ runStateT m s
```

```haskell
instance Monad m => Monad (StateT s m) where
  return = pure
  m >>= k = StateT $ \s -> do
    (a,t) <- runStateT m s
    runStateT (k a) t
```

MonadState

```haskell
class MonadState s m | m -> s where
  get :: m s
  put :: s -> m ()

modify :: (Monad m, MonadState s m) => (s -> s) -> m ()
modify f = do
  x <- get
  put (f x)

instance Applicative m => MonadState s (StateT s m) where
  get = StateT $ \s -> pure (s,s)
  put s = StateT $ \_ -> pure ((),s)
```

running StateT

```haskell
counterSI :: StateT Int Identity Int
counterSI = do
  x <- get
  modify (+1)
  return x

runCounterSI :: (Int,Int)
runCounterSI = runIdentity $ runStateT counterSI 0
```

```
-- (0,1)
```

MaybeT

```haskell
newtype MaybeT m a = MaybeT { runMaybeT :: m (Maybe a) }

instance Functor m => Functor (MaybeT m) where
  fmap f = MaybeT . fmap (fmap f) . runMaybeT

instance (Functor m, Monad m) => Applicative (MaybeT m) where
  pure = MaybeT . return . Just
  mf <*> mx = MaybeT $ do
    mb_f <- runMaybeT mf
    case mb_f of
      Nothing -> return Nothing
      Just f -> do
        mb_x <- runMaybeT mx
        case mb_x of
          Nothing -> return Nothing
          Just x -> return (Just (f x))
```

MaybeT

```haskell
newtype MaybeT m a = MaybeT { runMaybeT :: m (Maybe a) }

instance Functor m => Functor (MaybeT m) where
  fmap f = MaybeT . fmap (fmap f) . runMaybeT
```

```haskell
instance Monad m => Monad (MaybeT m) where
  return = pure
  x >>= f = MaybeT $ do
    v <- runMaybeT x
    case v of
      Nothing -> return Nothing
      Just y -> runMaybeT (f y)
```

running StateT and MaybeT

```haskell
counterSMI :: StateT Int (MaybeT Identity) Int
counterSMI = do
  x <- get
  modify (+1)
  return x

runCounterSMI :: Maybe (Int,Int)
runCounterSMI = runIdentity $ runMaybeT $ runStateT counterSMI 0
```

```
-- Just (0,1)
```

Achtung bei Nothings

```haskell
nothingness :: StateT Int (MaybeT Identity) Int
nothingness = StateT $ \s -> MaybeT (Identity Nothing)

runCounterSMI_nothing :: Maybe (Int,Int)
runCounterSMI_nothing = runIdentity $ runMaybeT $
    runStateT (counterSMI >> nothingness) 0
```

```
-- Nothing -- wo ist der Zaehlerstand?
```

Generische Monadische “Stacks”

```
counterGeneric :: (Monad m, MonadState Int m) => m Int
counterGeneric = do
   x <- get
   modify (+1)
   return x

runCounterGSMI :: Maybe (Int,Int)
runCounterGSMI = runIdentity $ runMaybeT $ runStateT
    counterGeneric 0
```

```
-- Just (0,1)
```

“Korrekte” Ordnung von Stacks beachten

```haskell
liftMaybeT :: Functor m => m a -> MaybeT m a
liftMaybeT = MaybeT . fmap Just

-- {-# LANGUAGE UndecidableInstances #-}
instance (Functor m, MonadState s m) => MonadState s (MaybeT m) where
  get = liftMaybeT get
  put = liftMaybeT . put

runCounterGMSI = runIdentity $ runStateT (runMaybeT counterGeneric) 0
```

```
-- (Just 0,1)
```