Monad transformers

Advanced functional programming - Lecture 4

Wouter Swierstra

Labs

I'll send you an example solution to the lab assignments.

All the submitted labs are now in the Github repository (in the `assignments`) directory.

Please mark your own lab exercise and write a brief `reflection.txt` on how you assessed your own work.

Also mark the *next* student in the list of submissions; add a `results.txt` file to their directory.

Create a pull request with both these documents and email me both marks before next week Tuesday.

Project

Good to see many people looking at the code.

Check out the `ants.pdf` for a project description.

The low-level ants instructions make it hard to write complex AI.

But perhaps a more high-level DSL can be compiled into such instructions…

Combining monads

- Applicative functors are closed under composition: if `f` and `g` are applicative, so is `f . g`.
- Monads, however, are **not** closed under such compositions.
- Can we define some other way to compose monads?

“List of successes” parsers

We have seen (applicative) parsers – but what about their monadic interface?

```
newtype Parser s a =
  Parser {runParser :: [s] -> [(a,[s])]
```

Question

How can we define a monad instance for such parsers?

Parser monad

```
instance Monad (Parser s) where
  return x = Parser (\xs -> [(x,xs)])
  p >>= f =
    Parser (\xs -> do (r,ys) <- runParser p xs
                      runParser (f r) ys)
```

This combines bot the state and list monads that we saw previously.

Question

From which instance is the >>= which is used in the do-expression taken?

Parser monad

```
instance Monad (Parser s) where
  return x = Parser (\xs -> [(x,xs)])
  p >>= f =
    Parser (\xs -> do (r,ys) <- runParser p xs
                      runParser (f r) ys)
```

This combines bot the state and list monads that we saw previously.

Question

From which instance is the >>= which is used in the do-expression taken?

Answer: `instance Monad []`

Monad transformers

We can actually assemble the parser monad from two building blocks: a list monad, and a state monad transformer.

```
newtype Parser s a =
  Parser  { runParser :: [s] -> [(a, [s])] }

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

Modulo wrapper types `StateT [s] [] a` is the same as

`[s] -> [(a, [s])]`.

Question

What is the kind of `StateT`?

Monad transformers (contd.)

```
instance (Monad m) => Monad (StateT s m) where
  return a  = StateT (\s -> return (a, s))
  m >>= f   = StateT
                (\s -> do (a, s') <- runStateT m s
                          runStateT (f a) s')
```

The instance definition is using the underlying monad `m` in the `do`-expression.

Monad transformers (contd.)

For (nearly) any monad, we can define a corresponding monad transformer, for instance:

```
newtype ListT m a =
  ListT  { runListT :: m [a] }
```

Monad transformers (contd.)

For (nearly) any monad, we can define a corresponding monad transformer, for instance:

```
newtype ListT m a =
  ListT  { runListT :: m [a] }

instance (Monad m) => Monad (ListT m) where
  return a    =  ListT (return [a])
  m >>= f     =
    ListT (do as   <- runListT m
              bss  <- mapM (runListT . f) as
              return (concat bbs))
```

Question:

Is `ListT (State s)` the same as `StateT s []`?

Order matters!

```
StateT s [] a
```

is

```
s -> [(a, s)]
```

whereas

```
ListT (State s) a
```

is

```
s -> ([a], s)
```

- Different orders of applying monads and monad transformers create subtly different monads!
- In the former monad, the new state depends on the result we select. In the latter, it doesn't.

Building blocks

- In order to see how to assemble monads from special-purpose monads, let us first learn about more monads than `Maybe`, `State`, `List` and `IO`.
- The place in the standard libraries for monads is `Control.Monad.*`.
- The state monad is available in `Control.Monad.State`.
- The list monad is available in `Control.Monad.List`.

Error or Either

The `Error` monad is a variant of `Maybe` which is slightly more useful for actually handling exceptions:

```
class Error e where
  noMsg   ::  e -> m a
  strMsg  ::  String -> e

instance Error e => Monad (Either e) where
  return x              =  Right x
  (Left  e)  >>=  _     =  Left e
  (Right r)  >>=  k     =  k r
  fail msg              =  Left (strMsg msg)

instance Error String where
  noMsg   =   ""
  strMsg  =   id
```

Error monad interface

Like `State`, the `Error` monad has an interface, such that we can throw and catch exceptions without requiring a specific underlying datatype:

```
class (Monad m) =>
  MonadError e m | (m -> e) where
    throwError  ::  e                          -> m a
    catchError  ::  m a -> (e -> m a)  -> m a

instance (Error e) => MonadError e (Either e)
```

The constraint `m -> e` in the class declaration is a *functional dependency*. It places certain restrictions on the instances that can be defined for that class.

Excursion: functional dependencies

- Type classes are *open relations* on types.
- Each single-parameter type class implicitly defines the set of types belonging to that type class.
- Instance definitions corresponds to membership.
- There is no need to restrict type classes to only one parameter.
- All parameters can also have different kinds.

Excursion: functional dependencies (contd.)

- Using a type class in a polymorphic context can lead to an *unresolved overloading* error:

```
show . read
```

What instance of `show` and `read` should be used?

Excursion: functional dependencies

- Multiple parameters lead to more unresolved overloading:

```
someComputation :: Either String Int
fallback :: Int
catchError someComputation (const (return fallback))
  :: (MonadError e (Either String)) =>
     Either String Int
```

The ‘handler’ doesn’t give any information about what the type of the errors is.

Excursion: functional dependencies (contd.)

- A functional dependency (inspired by relational databases) prevents such unresolved overloading.
- The dependency `m -> e` indicates that `e` is uniquely determined by `m`. The compiler can then automatically reduce a constraint such as

  ```
  (MonadError e (Either String)) => ...
  ```

  using

  ```
  instance (Error e) => MonadError e (Either e)
  ```

- Instance declarations that violate the functional dependency are rejected.

ErrorT monad transformer

Of course, there also is a monad transformer for errors:

```
newtype ErrorT e m a =
  ErrorT  { runErrorT :: m (Either e a) }

instance (Monad m, Error e) => Monad (ErrorT e m)
```

New combinations are possible. Even multiple transformers can be applied

Examples

```
ErrorT e (StateT s IO) a -- is the same as
StateT s IO (Either e a) -- is the same as
s -> IO (Either e a, s)

StateT s (ErrorT e IO) a   -- is the same as
s -> ErrorT e IO (a, s)    -- is the same as
s -> IO (Either e (a, s))
```

Question

Does an exception change the state or not? Can the resulting monad use `get`, `put`, `throwError`, `catchError`?

Defining interfaces

Many monads can have a state-like interface, hence we define:

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

    state f = do  s <- get
                  let ~(a, s') = f s
                  put s'
                  return a
```

Lifting

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

Lifting

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

instance (Error e) => MonadTrans (ErrorT e) where
   lift m = ErrorT (do  a <- m
                        return (Right a))

instance MonadTrans (StateT s) where
   lift m = StateT (\ s -> do  a <- m
                               return (a, s))
```

Lifting

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

instance (Error e) => MonadTrans (ErrorT e) where
   lift m = ErrorT (do  a <- m
                        return (Right a))

instance MonadTrans (StateT s) where
   lift m = StateT (\ s -> do  a <- m
                               return (a, s))

instance  (Error e, MonadState s m) =>
          MonadState s (ErrorT e m) where
   get  =   lift get
   put  =   lift . put
```

Question

How many instances are required?

A tour of Haskell's monads

Identity

The identity monad has no effects.

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

Question:

How is this a monad?

Identity

The identity monad has no effects.

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

Question:

How is this a monad?

```
instance Monad Identity where
    return x  =  Identity x
    m >>= f   =  Identity (f (runIdentity m))
```

Reader

The reader monad propagates some information, but unlike a state monad does not thread it through subsequent actions.

```
newtype Reader r a =
  Reader  { runReader :: r -> a }

instance Monad (Reader r) where
  return x  =  Reader (\r -> a)
  m >>= f   =  Reader (\r ->
                 runReader (f (runReader m r)) r)
```

Interface

We can also capture the interface of the operations that the reader monad supports:

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

Writer

The writer monad collects some information, but it is not possible to access the information already collected in previous computations.

```
newtype Writer w a =
  Writer { runWriter :: (a, w) }
```

To collect information, we have to know

- what an empty piece of information is, and
- how to combine two pieces of information.

A typical example is a list of things (`[]` and `(++)`), but the library generalizes this to any *monoid*.

Writer (contd.)

```
instance (Monoid w) => Monad (Writer w) where
  return x  =  Writer (x, mempty)
  m >>= f   = Writer $
    let  (a, w)   = runWriter m
         (b, w')  = runWriter (f a)
    in (b, w `mappend` w'))
```

Writer Interface

```
class  (Monoid w, Monad m) =>
       MonadWriter w m | m -> w where
  tell    ::  w -> m ()
  listen  ::  m a -> m (a, w)
  pass    ::  m (a, w -> w) -> m a
```

Cont

The continuation monad allows to capture the current continuation and jump to it when desired.

```
newtype Cont r a =
  Cont { runCont :: (a -> r) -> r }
```

Question

How is this a monad?

Cont

The continuation monad allows to capture the current continuation and jump to it when desired.

```
newtype Cont r a =
  Cont { runCont :: (a -> r) -> r }
```

Question

How is this a monad?

```
instance Monad (Cont r) where
  return a  =  Cont (\ k -> k a)
  m >>= f   =  Cont
     (\ k -> runCont m (\ a -> runCont (f a) k))
```

MonadPlus

```
class (Monad m) => MonadPlus m where
  mzero  ::  m a
  mplus  ::  m a -> m a -> m a

instance MonadPlus [] where
  mzero  =   []
  mplus  =   (++)

instance MonadPlus Maybe where
  mzero  =   Nothing

  Nothing  `mplus`  ys  =  ys
  xs       `mplus`  ys  =  xs

msum   ::   MonadPlus m => [m a] -> m a
guard  ::   MonadPlus m => Bool -> m ()
```

Recap: Monad transformers

Monad transformers allow you to assemble complex monads in a structured fashion.

The do **not** commute.

Lifting various operations through stacks of monad transformers can be cumbersome.

There is recent interest in an alternative approach: algebraic effects.

The idea is to separate the syntax from computations from their semantics.

We use various monadic operations (such as `get` or `throw`) and only later decide on the order that we want to stack the corresponding monad transformers.

Summary

- Common interfaces are extremely powerful and give you a huge amount of predefined theory and functions.
- Look for common interfaces in your programs.
- Recognise monads and applicative functors in your programs.
- Define or assemble your own monads.
- Add new features to the monads you are using.
- Monads and applicative functors make Haskell particularly suited for Embedded Domain Specific Languages.