Advanced Functional Programming

2012-2013, periode 2

Doaitse Swierstra

Department of Information and Computing Sciences
Utrecht University

Nov 22, 2010

5. Monad transformers

Combining monads

- A strong point of monads is that different monads can be combined into new monads.
- If monadic code does not exploit the implementation of its underlying implementation directly (i.e., if a state modifier only uses get and put), the monad underlying a specific bit of code can be changed to deal with new kinds of effects.

Parsers

- The so called "list-of-successes" type of parsers is a monad:

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

- We have a combination of a state and a list monad.

```
instance Monad (Parser s) where
   return x = Parser (λxs → [(x, xs)])
   p ≫= f   = Parser (λxs → do
                              (r, ys) ← runParser p xs
                              runParser (f r) ys)
```

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)}
StateT [s] [] a ≈ [s] → [(a, [s])]
```

Question

What is the kind of StateT?

Monad transformers (contd.)

```
instance (Monad m) ⇒ Monad (StateT s m) where
  return a = StateT (λs → return (a, s))
  m ≫= f  = StateT (λs → do (a, s′) ← runStateT m s
                            runStateT (f a) s′)
```

The instance definition is using the underlying monad.

Monad transformers (contd.)

```
instance (Monad m) ⇒ Monad (StateT s m) where
   return a = StateT (λs → return (a, s))
   m ≫= f  = StateT (λs → do (a, s′) ← runStateT m s
                               runStateT (f a) s′)
```

The instance definition is using the underlying monad. Even more explicitly, using the underlying ≫=:

```
m ≫= f = StateT (λs → runStateT m s ≫= (λ(a, s′)
                     → runStateT (f a) s′)
```

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 a ← runListT m
                        b ← mapM (runListT ∘ f) a
                        return (concat b)
```

Question

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

Order matters!

$$\begin{aligned}\mathsf{StateT\ s\ [\,]\ a} &\approx \mathsf{s} \to [(\mathsf{a}, \mathsf{s})] \\ \mathsf{ListT\ (State\ s)\ a} &\approx \mathsf{s} \to ([\mathsf{a}], \mathsf{s})\end{aligned}$$

Order matters!

$$\begin{aligned}
\mathsf{StateT\ s\ []\ a} &\approx \mathsf{s} \rightarrow [(\mathsf{a}, \mathsf{s})] \\
\mathsf{ListT\ (State\ s)\ a} &\approx \mathsf{s} \rightarrow ([\mathsf{a}], \mathsf{s})
\end{aligned}$$

- 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.

5.1 More monads

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 avilable 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
  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 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:

  $$\mathsf{show} \circ \mathsf{read} :: (\mathsf{Read}\ \mathsf{a}) \Rightarrow \mathsf{String} \rightarrow \mathsf{String}$$

  Variables in the constraint no longer occur in the type.

Excursion: functional dependencies (contd.)

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

```
show ∘ read :: (Read a) ⇒ String → String
```

  Variables in the constraint no longer occur in the type.

- Multiple parameters lead to more unresolved overloading:

```
class (Monad m) ⇒ MonadError e m | m → e where
  throwError :: e → m a
  catchError :: m a → (e → m a) → m a
someComputation :: Either String Int
fallback :: Int
catchError someComputation (const (return fallback))
   :: (MonadError e (Either String)) ⇒ Either String Int
```

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:

```
ErrorT e (StateT s IO) a
   ≈ StateT s IO (Either e a)
   ≈ s → IO (Either e a, s)
```

```
StateT s (ErrorT e IO) a
   ≈ s → ErrorT e IO (a, s)
   ≈ s → IO (Either e (a, s))
```

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

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 (λs → do a ← m
                           return (a, s))
instance (Error e, MonadState s m) ⇒
            MonadState s (ErrorT e m) where
  get = lift get
  put = lift ∘ put
```

How many instances are required?

Identity

The identity monad has no effects.

```
newtype Identity a = Identity {runIdentity :: a}
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 a = Reader (λr → a)
   m ≫= f  = Reader (λr → runReader (f (runReader m r)) r)
```

Interface:

```
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.

Monoids

Monoids are algebraic structures (defined in Data.Monoid) with a neutral element and an associative binary operation:

```
class Monoid a where
  mempty  :: a
  mappend :: a → a → a
  mconcat :: [a] → a
  mconcat = foldr mappend mempty
instance Monoid [a] where
  mempty  = [ ]
  mappend = (⧺)
```

. . . and many more! Note the similarity to monads!

Writer (contd.)

```
instance (Monoid w) ⇒ Monad (Writer w) where
   return a = Writer (a, mempty)
   m ≫= f  = Writer (let (a, w)  = runWriter m
                         (b, w′) = runWriter (f a)
                     in (b, w ‘mappend‘ w′))
```

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}
instance Monad (Cont r) where
  return a = Cont (λk → k a)
  m ≫= f  = Cont (λk → runCont m (λa → runCont (f a) k))
```

Interface:

```
instance MonadCont (Cont r) where
  callCC f =
    Cont (λk → runCont (f (λa → Cont (λ_ → k a))) k)
```

Continuation example

Implementing a C-style for-loop with break and continue:

```
type CIO r a = ContT r IO a

for :: (Int, Int → Bool, Int → Int) →
        (CIO r s → CIO r t → Int → CIO r ()) → CIO r ()
for (i, c, s) body
    | c i = callCC (λbreak → callCC (λcontinue →
                body (break ()) (continue ()) i) ≫ for (s i, c, s) body)
    | otherwise = return ()

main = runContT main′ return

main′ :: CIO r ()
main′ = for (0, const True, (+1))
               (λbreak continue i →
                  do when (even i)  continue
                     when (i ⩾ 12) break
                     lift $ putStrLn $ "iteration " ++ show i)
```

5.2 Related structures

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 ()
```

Applicative **(applicative functors)**

```
class (Functor f) ⇒ Applicative f where
   pure  :: a → f a
   (<∗>) :: f (a → b) → f a → f b
```

The (<∗>) operation is like ap:

```
ap :: (Monad m) ⇒ m (a → b) → m a → m b
```

Every functor supports map:

```
(<$>) :: Functor f ⇒ (a → b) → f a → f b
```

- Note the parser interface!
- Easy to see: every monad is an applicative functor/idiom.
- But not every applicative functor is a monad.

Monads vs. applicative functors, informally

```
(<∗>) :: (Applicative f) ⇒ f (a →     b) → f a  → f b
(=≪)  :: (Monad m)       ⇒   (a → m b) → m a → m b
```

- Intuitively, applicative functors don't dictate a full sequencing of effects.
- With monads, subsequent actions can depend on the results of effects.
- With applicative functors, the structure is statically determined (and can be analyzed or optimized).

Example: lists

We can impose a different applicative functor structure on lists from that induced via the list monad:

```
pure x = repeat x
(f : fs) <*> (x : xs) = f x : (fs <*> xs)
_        <*> _        = []
```

Note that f <$> xs = pure x <*> xs.

With these functions, we can define transpose as follows:

```
transpose :: [[a]] → [[a]]
transpose []         = pure []
transpose (xs : xss) = (:) <$> xs <*> transpose xss
```

Example: Failure

```
instance (Monoid e) ⇒ Applicative (Either e) where
   pure x = Right x
   Right f  <∗> Right x  = Right (f x)
   Left e1  <∗> Left e2  = Left (e1 ‘mappend‘ e2)
   Left e1  <∗> Right _  = Left e1
   Right _  <∗> Left e2  = Left e2
```

This definition is different from the error monad in that multiple failures are collected!

Applicative functor laws

- identity

```
pure id <*> u = u
```

- composition

```
pure (∘) <*> u <*> v <*> w = u <*> (v <*> w)
```

- homomorphism

```
pure f <*> pure x = pure (f x)
```

- interchange

```
u <*> pure x = pure (λf → f x) <*> u
```

Proposed applicative functor notation

Most applicative functor operations take the form

> $\text{pure } f \mathbin{<\!\ast\!>} x_1 \mathbin{<\!\ast\!>} \ldots \mathbin{<\!\ast\!>} x_n$
>
> $f \mathbin{<\!\$\!>} x_1 \mathbin{<\!\ast\!>} \ldots \mathbin{<\!\ast\!>} x_n$

McBride and Paterson propose to write this as

> $[\![\, f\ x_1 \ldots x_n \,]\!]$

More on applicative functors

- Lots of derived functions, for instance for traversing structures.
- The composition of two applicative functors is always an applicative functor again, and this can easily be expressed in Haskell code.

Arrows

```
class Arrow a where
  arr   :: (b → c) → a b c
  (⋙) :: a b c → a c d → a b d
  first :: a b c → a (b, d) (c, d)
```

- Every monad can be made into an arrow.
- Every arrow can be made into an applicative functor.
- Arrows turn out to require a complex set of additional classes that add additional operations, and have a rather complicated associated syntax proposal.

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.
- Monads (Wadler, Moggi) are stronger than applicative functors. Applicative functors (McBride, Paterson) are more flexible. Arrows (Hughes) are yet another alternative.
- Monads have proved themselves. Time will tell whether Applicative functors or arrows can be equally successful.