

Advanced Functional Programming 2011-2012, period 2

Andres Löh and Doaitse Swierstra

Department of Information and Computing Sciences
Utrecht University

December 8, 2011

8. 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.



Faculty of Science

Parsers

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

```
\label{eq:parser} \begin{array}{l} \textbf{newtype} \ \mathsf{Parser} \ \mathsf{s} \ \mathsf{a} = \\ \mathsf{Parser} \ \{\mathsf{runParser} :: [\mathsf{s}] \to [(\mathsf{a},[\mathsf{s}])] \} \end{array}
```

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

```
\label{eq:parser_solution} \begin{array}{l} \textbf{instance} \ \ \textbf{Monad} \ \ (\mathsf{Parser} \ s) \ \textbf{where} \\ \text{return } x = \mathsf{Parser} \ (\lambda \mathsf{xs} \to [(\mathsf{x}, \mathsf{xs})]) \\ \text{p} \ggg \mathsf{f} \ = \mathsf{Parser} \ (\lambda \mathsf{xs} \to \textbf{do} \\ & (\mathsf{r}, \mathsf{ys}) \leftarrow \mathsf{runParser} \ \mathsf{p} \ \mathsf{xs} \\ & \mathsf{runParser} \ (\mathsf{f} \ \mathsf{r}) \ \mathsf{ys}) \end{array}
```

Monad transformers

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

```
\label{eq:newtype} \begin{split} & \text{newtype Parser s a} = \\ & \text{Parser } \{\text{runParser :: } [s] \rightarrow [(a,[s])] \} \\ & \text{newtype StateT s m a} = \\ & \text{StateT } \{\text{runStateT :: } s \rightarrow \text{m } (a,s) \} \\ & \text{StateT } [s] \ [] \ a \approx [s] \rightarrow [(a,[s])] \end{split}
```

Question

What is the kind of StateT?



Monad transformers (contd.)

```
\label{eq:monad_m} \begin{split} & \text{instance } (\mathsf{Monad}\ \mathsf{m}) \Rightarrow \mathsf{Monad}\ (\mathsf{StateT}\ \mathsf{s}\ \mathsf{m})\ \text{where} \\ & \mathsf{return}\ \mathsf{a} = \mathsf{StateT}\ (\lambda \mathsf{s} \to \mathsf{return}\ (\mathsf{a},\mathsf{s})) \\ & \mathsf{m} \ggg \mathsf{f}\ = \mathsf{StateT}\ (\lambda \mathsf{s} \to \textcolor{red}{\textbf{do}}\ (\mathsf{a},\mathsf{s}') \leftarrow \mathsf{runStateT}\ \mathsf{m}\ \mathsf{s} \\ & \mathsf{runStateT}\ (\mathsf{f}\ \mathsf{a})\ \mathsf{s}') \end{split}
```

The instance definition is using the underlying monad.

Faculty of Science

Monad transformers (contd.)

```
\begin{array}{l} \textbf{instance} \; (\mathsf{Monad} \; \mathsf{m}) \Rightarrow \mathsf{Monad} \; (\mathsf{StateT} \; \mathsf{s} \; \mathsf{m}) \; \textbf{where} \\ \mathsf{return} \; \mathsf{a} = \mathsf{StateT} \; (\lambda \mathsf{s} \rightarrow \mathsf{return} \; (\mathsf{a}, \mathsf{s})) \\ \mathsf{m} \gg \mathsf{f} \; = \mathsf{StateT} \; (\lambda \mathsf{s} \rightarrow \textbf{do} \; (\mathsf{a}, \mathsf{s}') \leftarrow \mathsf{runStateT} \; \mathsf{m} \; \mathsf{s} \\ \mathsf{runStateT} \; (\mathsf{f} \; \mathsf{a}) \; \mathsf{s}') \end{array}
```

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

$$\label{eq:mass} \begin{split} \mathsf{m} \ggg \mathsf{f} &= \mathsf{StateT} \ (\lambda \mathsf{s} \to \mathsf{runStateT} \ \mathsf{m} \ \mathsf{s} \ggg (\lambda(\mathsf{a},\mathsf{s}') \\ &\to \mathsf{runStateT} \ (\mathsf{f} \ \mathsf{a}) \ \mathsf{s}' \end{split}$$

Monad transformers (contd.)

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

```
\label{eq:listT} \begin{array}{l} \textbf{newtype} \ \mathsf{ListT} \ \{\mathsf{runListT} :: \mathsf{m} \ [\mathsf{a}] \} \\ \\ \textbf{instance} \ (\mathsf{Monad} \ \mathsf{m}) \Rightarrow \mathsf{Monad} \ (\mathsf{ListT} \ \mathsf{m}) \ \textbf{where} \\ \\ \mathsf{return} \ \mathsf{a} = \mathsf{ListT} \ (\mathsf{return} \ [\mathsf{a}]) \\ \\ \mathsf{m} \gg \mathsf{f} \ = \mathsf{ListT} \ (\textbf{do} \ \mathsf{a} \leftarrow \mathsf{runListT} \ \mathsf{m} \\ \\ \mathsf{b} \leftarrow \mathsf{mapM} \ (\mathsf{runListT} \circ \mathsf{f}) \ \mathsf{a} \\ \\ \\ \mathsf{return} \ (\mathsf{concat} \ \mathsf{b}) \end{array}
```

Question

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



Order matters!

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



Faculty of Science

Order matters!

StateT s [] a
$$\approx$$
 s \rightarrow [(a, s)]
ListT (State s) a \approx s \rightarrow ([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.



8.1 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.



[Faculty of Science

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 \rightarrow e
instance Error e \Rightarrow Monad (Either e) where
  return x = Right x
  (Left e) \gg  _ = Left e
  (Right r) \gg 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:

The constraint $m \to e$ 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.



[Faculty of Science

Excursion: functional dependencies (contd.)

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

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

Variables in the constraint no longer occur in the type.

Faculty of Science

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} \to \mathsf{String}
```

Variables in the constraint no longer occur in the type.

▶ Multiple parameters lead to more unresolved overloading:

```
 \begin{array}{l} \textbf{class} \; (\mathsf{Monad} \; \mathsf{m}) \Rightarrow \mathsf{MonadError} \; \mathsf{e} \; \mathsf{m} \; | \; \mathsf{m} \rightarrow \mathsf{e} \; \textbf{where} \\ \; \mathsf{throwError} :: \mathsf{e} \rightarrow \mathsf{m} \; \mathsf{a} \\ \; \mathsf{catchError} :: \mathsf{m} \; \mathsf{a} \rightarrow (\mathsf{e} \rightarrow \mathsf{m} \; \mathsf{a}) \rightarrow \mathsf{m} \; \mathsf{a} \\ \; \mathsf{someComputation} :: \mathsf{Either} \; \mathsf{String} \; \mathsf{Int} \\ \; \mathsf{fallback} :: \mathsf{Int} \\ \; \mathsf{catchError} \; \mathsf{someComputation} \; (\mathsf{const} \; (\mathsf{return} \; \mathsf{fallback})) \\ \; :: (\mathsf{MonadError} \; \mathsf{e} \; (\mathsf{Either} \; \mathsf{String})) \Rightarrow \mathsf{Either} \; \mathsf{String} \; \mathsf{Int} \\ \end{aligned}
```

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

```
(\mathsf{MonadError}\;\mathsf{e}\;(\mathsf{Either}\;\mathsf{String})) \Rightarrow \dots using
```

- **instance** (Error e) \Rightarrow 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:

```
\label{eq:mewtype} \begin{split} & \textbf{newtype} \; \mathsf{ErrorT} \; \mathsf{e} \; \mathsf{m} \; \mathsf{a} = \\ & \mathsf{ErrorT} \; \{\mathsf{runErrorT} :: \mathsf{m} \; (\mathsf{Either} \; \mathsf{e} \; \mathsf{a}) \} \\ & \mathsf{instance} \; (\mathsf{Monad} \; \mathsf{m}, \mathsf{Error} \; \mathsf{e}) \Rightarrow \mathsf{Monad} \; (\mathsf{ErrorT} \; \mathsf{e} \; \mathsf{m}) \end{split}
```

New combinations are possible. Even multiple transformers can be applied:

```
ErrorT e (StateT s IO) a
   \approx StateT s IO (Either e a)
   \approx s \rightarrow IO (Either e a, s)
```

```
StateT s (ErrorT e IO) a
\approx s \rightarrow \text{ErrorT e IO } (a, s)
\approx s \rightarrow \text{IO } (\text{Either e } (a, s))
```

Does an exception change the state or not? Can the resulting

monad use get, put, throwError, catchError?

[Faculty of Science Information and Computing Sciences

Universiteit Utrecht

◆□▶◆御▶◆団▶◆団▶ 団 めの◎

Lifting

```
class MonadTrans t where
   \mathsf{lift} :: \mathsf{Monad} \; \mathsf{m} \Rightarrow \mathsf{m} \; \mathsf{a} \to \mathsf{t} \; \mathsf{m} \; \mathsf{a}
instance (Error e) ⇒ MonadTrans (ErrorT e) where
   lift m = ErrorT (do a \leftarrow m)
                               return (Right a))
instance MonadTrans (StateTs) where
   lift m = StateT (\lambdas \rightarrow do a \leftarrow m
                                       return (a, s))
instance (Error e, MonadState s m) \Rightarrow
             MonadState s (ErrorT e m) where
   get = lift get
   put = lift \circ 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.

```
\label{eq:newtype} \begin{split} & \textbf{newtype} \ \text{Reader} \ r \ a = \text{Reader} \ \{ \text{runReader} :: r \to a \} \\ & \textbf{instance} \ \text{Monad} \ (\text{Reader} \ r) \ \textbf{where} \\ & \text{return} \ a = \text{Reader} \ (\lambda r \to a) \\ & \text{m} \ggg f \ = \text{Reader} \ (\lambda r \to \text{runReader} \ (f \ (\text{runReader} \ m \ r)) \ r) \end{split}
```

Interface:

Writer

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

```
\textbf{newtype} \ \mathsf{Writer} \ \mathsf{w} \ \mathsf{a} = \mathsf{Writer} \ \{ \mathsf{runWriter} :: (\mathsf{a}, \mathsf{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.

◆□▶◆御▶◆団▶◆団▶ 団 めの◎

Monoids

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

```
class Monoid a where mempty :: a \rightarrow a \rightarrow a \rightarrow a mconcat :: [a] \rightarrow a mconcat = foldr mappend mempty instance Monoid [a] where mempty = [] mappend = (++)
```

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



Writer (contd.)

Interface:

```
 \begin{array}{c} \textbf{class} \; (\mathsf{Monoid} \; \mathsf{w}, \mathsf{Monad} \; \mathsf{m}) \Rightarrow \\ & \; \mathsf{MonadWriter} \; \mathsf{w} \; \mathsf{m} \; | \; \mathsf{m} \to \mathsf{w} \; \textbf{where} \\ \mathsf{tell} \; \; :: \mathsf{w} \to \mathsf{m} \; () \\ \mathsf{listen} :: \mathsf{m} \; \mathsf{a} \to \mathsf{m} \; (\mathsf{a}, \mathsf{w}) \\ \mathsf{pass} \; :: \mathsf{m} \; (\mathsf{a}, \mathsf{w} \to \mathsf{w}) \to \mathsf{m} \; \mathsf{a} \\ \end{array}
```

Cont

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

```
\label{eq:cont_rate} \begin{split} & \textbf{newtype} \; \mathsf{Cont} \; r \; a \Rightarrow \mathsf{Cont} \; \{ \mathsf{runCont} :: (\mathsf{a} \to \mathsf{r}) \to \mathsf{r} \} \\ & \textbf{instance} \; \mathsf{Monad} \; (\mathsf{Cont} \; \mathsf{r}) \; \textbf{where} \\ & \mathsf{return} \; \mathsf{a} = \mathsf{Cont} \; (\lambda \mathsf{k} \to \mathsf{k} \; \mathsf{a}) \\ & \mathsf{m} \not \gg \mathsf{f} \; = \mathsf{Cont} \; (\lambda \mathsf{k} \to \mathsf{runCont} \; \mathsf{m} \; (\lambda \mathsf{a} \to \mathsf{runCont} \; (\mathsf{f} \; \mathsf{a}) \; \mathsf{k})) \end{split}
```

Interface:

```
\label{eq:cont_cont} \begin{array}{l} \textbf{instance} \ \mathsf{MonadCont} \ (\mathsf{Cont} \ r) \ \textbf{where} \\ \mathsf{callCC} \ f = \\ \mathsf{Cont} \ (\lambda \mathsf{k} \to \mathsf{runCont} \ (\mathsf{f} \ (\lambda \mathsf{a} \to \mathsf{Cont} \ (\lambda_- \to \mathsf{k} \ \mathsf{a}))) \ \mathsf{k}) \end{array}
```

Continuation example

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

```
\label{eq:type_cont} \begin{split} \textbf{type} & \; \mathsf{CIO} \; \mathsf{r} \; \mathsf{a} = \mathsf{ContT} \; \mathsf{r} \; \mathsf{IO} \; \mathsf{a} \\ \mathsf{for} :: (\mathsf{Int}, \mathsf{Int} \to \mathsf{Bool}, \mathsf{Int} \to \mathsf{Int}) \to \\ & \; (\mathsf{CIO} \; \mathsf{r} \; \mathsf{s} \to \mathsf{CIO} \; \mathsf{r} \; \mathsf{t} \to \mathsf{Int} \to \mathsf{CIO} \; \mathsf{r} \; ()) \to \mathsf{CIO} \; \mathsf{r} \; () \end{split}
 for (i, c, s) body
         | c i = callCC (\lambda break \rightarrow callCC (\lambda continue \rightarrow
                            body (break ()) (continue ()) i) \gg for (s i, c, s) body)
         otherwise = return ()
 main = runContT main' return
 main' :: CIO r ()
 main' = for (0, const True, (+1))
                            (\lambda break continue i \rightarrow
                                 do when (even i) continue
                                        when (i \ge 12) break
                                        lift $ putStrLn $ "iteration " ++ show i)
```

◆□▶◆御▶◆団▶◆団▶ 団 めの◎

8.2 Related structures





MonadPlus

```
class (Monad m) ⇒ MonadPlus m where
   mplus :: m a \rightarrow m a \rightarrow m a
instance MonadPlus [] where
   mzero = []
   mplus = (++)
instance MonadPlus Maybe where
   mzero = Nothing
   Nothing 'mplus' ys = ys
            'mplus' ys = xs
   XS
\mathsf{msum} :: \mathsf{MonadPlus} \; \mathsf{m} \Rightarrow [\mathsf{m} \; \mathsf{a}] \to \mathsf{m} \; \mathsf{a}
guard :: MonadPLus m \Rightarrow Bool \rightarrow m ()
```



Applicative (applicative functors)

The (<*>) operation is like ap:

$$\mathsf{ap} :: (\mathsf{Monad}\;\mathsf{m}) \Rightarrow \mathsf{m}\; (\mathsf{a} \to \mathsf{b}) \to \mathsf{m}\; \mathsf{a} \to \mathsf{m}\; \mathsf{b}$$

Every functor supports map:

$$(\$$
 $)$:: Functor $f \Rightarrow (a \rightarrow b) \rightarrow f \ a \rightarrow f \ b$

- ▶ 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

$$(<*>) :: (Applicative f) \Rightarrow f (a \rightarrow b) \rightarrow f a \rightarrow f b$$
$$(=\ll) :: (Monad m) \Rightarrow (a \rightarrow m b) \rightarrow m a \rightarrow m b$$

- 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:

Note that $f \le xs = pure x \le xs$.

With these functions, we can define transpose as follows:

```
 \begin{array}{l} transpose :: [[a]] \rightarrow [[a]] \\ transpose [] &= pure [] \\ transpose (xs : xss) = (:) <\$> xs <*> transpose xss \\ \end{array}
```

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!

4日 > 4 個 > 4 豆 > 4 豆 > 豆 めの()

Applicative functor laws

▶ identity

pure id
$$<*>$$
 u = u

composition

pure
$$(\circ)$$
 <*> u <*> v <*> w = u <*> $(v <*> w)$

► homomorphism

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

interchange

u <*> pure x = pure
$$(\lambda f \rightarrow f x)$$
 <*> u

Proposed applicative functor notation

Most applicative functor operations take the form

pure f <*>
$$x_1$$
 <*> ... <*> x_n f <\$> x_1 <*> ... <*> x_n

McBride and Paterson propose to write this as

$$\llbracket f x_1 \dots x_n \rrbracket$$

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.



Faculty of Science

```
class Arrow a where  \begin{array}{ll} \mathsf{arr} & :: (\mathsf{b} \to \mathsf{c}) \to \mathsf{a} \; \mathsf{b} \; \mathsf{c} \\ (\ggg) :: \mathsf{a} \; \mathsf{b} \; \mathsf{c} \to \mathsf{a} \; \mathsf{c} \; \mathsf{d} \to \mathsf{a} \; \mathsf{b} \; \mathsf{d} \\ \mathsf{first} & :: \mathsf{a} \; \mathsf{b} \; \mathsf{c} \to \mathsf{a} \; (\mathsf{b}, \mathsf{d}) \; (\mathsf{c}, \mathsf{d}) \end{array}
```

- 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.

◆□▶◆御▶◆団▶◆団▶ 団 めの◎