Programming with Monad Transformers

Iavor S. Diatchki

Galois Tech Seminar

What is a monad?

According to `www.googlism.com` a monad is...

What is a monad?

According to `www.googlism.com` a monad is...

- ...a card game?

What is a monad?

According to `www.googlism.com` a monad is...

- ...a card game?
- ...the highest spiritual aspect of the human being?

What is a monad?

According to `www.googlism.com` a monad is...

- ...a card game?
- ...the highest spiritual aspect of the human being?
- ...a very specific type of agreement?

What is a monad?

According to `www.googlism.com` a monad is...

- ...a card game?
- ...the highest spiritual aspect of the human being?
- ...a very specific type of agreement?
- ...capable of creating twelve souls?

What is a monad?

According to `www.googlism.com` a monad is...

- ...a card game?
- ...the highest spiritual aspect of the human being?
- ...a very specific type of agreement?
- ...capable of creating twelve souls?
- ...like a mirror?

What is a monad?

According to `www.googlism.com` a monad is...

- ...a card game?
- ...the highest spiritual aspect of the human being?
- ...a very specific type of agreement?
- ...capable of creating twelve souls?
- ...like a mirror?
- ...surrounded by a hazy glow?

Monads in Haskell

```
class Monad m where
  return :: a → m a
  (>>=)  :: m a → (a → m b) → m b
```

Monads in Haskell

```
class Monad m where
  return :: a → m a
  (>>=)  :: m a → (a → m b) → m b
```

- Example

```
ex1 :: (Monad m) ⇒ m Int
ex1  = do x ← return 1
          y ← return 2
          return (x + y)
```

Monads in Haskell

```
class Monad m where
  return :: a → m a
  (>>=)  :: m a → (a → m b) → m b
```

- Example

```
ex1 :: (Monad m) ⇒ m Int
ex1  = do x ← return 1
          y ← return 2
          return (x + y)
```

- Desugared to:

```
ex1 :: (Monad m) ⇒ m Int
ex1  = return 1 >>= λx →
       return 2 >>= λy →
       return (x + y)
```

Reader (Environment) Monads

- Monads that provide access to a context of type i.

```
class (Monad m) ⇒ ReaderM m i | m ⇝ i where
  ask :: m i      -- Get the context.
```

Reader (Environment) Monads

- Monads that provide access to a context of type i.

```
class (Monad m) ⇒ ReaderM m i | m ⇝ i where
  ask :: m i      -- Get the context.
```

- Example

```
type Env = [(String,String)]

lkp :: (ReaderM m Env) ⇒ String → m (Maybe (String))
lkp var = do env ← ask
             return (lookup var env)
```

Writer (Output) Monads

- Monads that can collect values of type i.

```
class (Monad m) ⇒ WriterM m i | m ⇝ i where
  put  :: i → m ()    -- Add values to the collection.
```

Writer (Output) Monads

- Monads that can collect values of type i.

```
class (Monad m) ⇒ WriterM m i | m ⇝ i where
  put  :: i → m ()    -- Add values to the collection.
```

- Example

```
data Tree = Node String [Tree] | Leaf String

leaves :: (WriterM m [String]) ⇒ Tree → m ()
leaves (Leaf x)      = put [x]
leaves (Node _ xs)   = mapM_ leaves xs
```

State Monads

- Monads that propagate a state component of type i.

```
class (Monad m) ⇒ StateM m i | m ⇝ i where
  get :: m i          -- Get the state.
  set :: i → m ()     -- Set the state.
```

State Monads

- Monads that propagate a state component of type i.

```
class (Monad m) ⇒ StateM m i | m ⇝ i where
  get :: m i          -- Get the state.
  set :: i → m ()     -- Set the state.
```

- Example

```
sum :: (StateM m Int) ⇒ [Int] → m Int
sum (x:xs)  = do tot ← get
                 set (tot + x)
                 sum xs
sum []      = get
```

Exception Monads

- Monads that support raising exceptions of type *i*.

```
class (Monad m) ⇒ ExceptionM m i | m ⇝ i where
  raise :: i → m a  -- Raise an exception.
```

Exception Monads

- Monads that support raising exceptions of type *i*.

```
class (Monad m) ⇒ ExceptionM m i | m ⇝ i where
  raise :: i → m a  -- Raise an exception.
```

- Example

```
divide :: (ExceptionM m String) ⇒ Int → Int → m Int
divide _ 0  = raise "Division by 0."
divide x y  = return (div x y)
```

Continuation Monads

- Monads that provide access to a computation's continuation.

```
class (Monad m) ⇒ ContM m where
  callCC :: ((a → m b) → m a) → m a
```

Continuation Monads

- Monads that provide access to a computation's continuation.

```
class (Monad m) ⇒ ContM m where
  callCC :: ((a → m b) → m a) → m a
```

- Example

```
ten :: (ContM m) ⇒ m () → m ()
ten m = do (x,l) ← labelCC 1
           m
           when (x ≤ 10) (jump (x+1) l)
```

Summary of Monadic Flavors

- We have categorized all monads according to the operations that they support:
  `ReaderM`, `WriterM`, `StateM`, `ExceptionM`, `ContM`

Summary of Monadic Flavors

- We have categorized all monads according to the operations that they support:

  `ReaderM`, `WriterM`, `StateM`, `ExceptionM`, `ContM`

- The type system keeps track of the effects that we use:

```
test :: (...) ⇒ m ()
test = do env ← ask
          x   ← get
          when (x > env) (raise "Too big!")
```

Summary of Monadic Flavors

- We have categorized all monads according to the operations that they support:

  `ReaderM`, `WriterM`, `StateM`, `ExceptionM`, `ContM`

- The type system keeps track of the effects that we use:

```
test :: (...) ⇒ m ()
test = do env ← ask
          x   ← get
          when (x > env) (raise "Too big!")
```

- The context is:

  `(ReaderM m a, StateM m a, ExceptionM m String, Ord a)`

Ice Cream Flavors

Implementing the API

- We need an implementation for the API that we defined.
- Some options:
  - Define one (big) type that supports all effects.
  - Define monads customized to a particular use.

We can use monad transformers to construct custom monads in a modular fashion.

Monad Transformers

- A *monad transformer* extends an existing monad with support for some additional effects:

```
class MonadT t where
  lift :: (Monad m) ⇒ m a → (t m) a
```

Monad Transformers

- A *monad transformer* extends an existing monad with support for some additional effects:

```
class MonadT t where
  lift :: (Monad m) ⇒ m a → (t m) a
```

- The library provides a family of transformers—one for each effect flavor:

`ReaderT`, `WriterT`, `StateT`, `ExceptionT`, `ContT`

Monad Transformers

- A *monad transformer* extends an existing monad with support for some additional effects:

```
class MonadT t where
  lift :: (Monad m) ⇒ m a → (t m) a
```

- The library provides a family of transformers—one for each effect flavor:

  `ReaderT`, `WriterT`, `StateT`, `ExceptionT`, `ContT`

- Example—adding exceptions and state to IO:

```
ExceptionT String (StateT Int IO)
```

Base Monads

- To construct a monad:
  - Start with an existing (base) monad, then
  - Use transformers to add support for additional effects

Base Monads

- To construct a monad:
  - Start with an existing (base) monad, then
  - Use transformers to add support for additional effects
- Accessing the base monad:

```
class (Monad m, Monad n) ⇒ BaseM m n | m ⇝ n where
  inBase :: n a → m a
```

- Example:

```
hello :: (ReaderM m Int, BaseM m IO) ⇒ m ()
hello  = do x ← ask
            inBase (print x)
```

- The library provides two base monads `Id` and `Lift`

Connecting the Implementation to the API

- For every transformer, the library provides instances that implement the monadic API.
- Each instance either:
  - implements the given effect, or
  - delegates the action to the underlying monad.

Connecting the Implementation to the API

- For every transformer, the library provides instances that implement the monadic API.
- Each instance either:
  - implements the given effect, or
  - delegates the action to the underlying monad.
- Example

```
ex :: ReaderT Int (StateT Int Id) Int
ex  = do x ← ask
         y ← get
         return (x+y)
```

A Monadic Tower

Execution

- So far:
  - We defined an API for effects,
  - Used transformers to define types that implement the API.
- We also need a way to execute computations.

Execution

- So far:
  - We defined an API for effects,
  - Used transformers to define types that implement the API.
- We also need a way to execute computations.
- Each transformer is equipped with a function that eliminates it:
  - `runReaderT :: i → ReaderT i m a → m a`

Execution

- So far:
  - We defined an API for effects,
  - Used transformers to define types that implement the API.
- We also need a way to execute computations.
- Each transformer is equipped with a function that eliminates it:
  - `runReaderT :: i → ReaderT i m a → m a`
  - `runWriterT :: WriterT i m a → m (a,i)`

Execution

- So far:
  - We defined an API for effects,
  - Used transformers to define types that implement the API.
- We also need a way to execute computations.
- Each transformer is equipped with a function that eliminates it:
  - `runReaderT :: i → ReaderT i m a → m a`
  - `runWriterT :: WriterT i m a → m (a,i)`
  - `runStateT :: i → StateT i m a → m (a,i)`

Execution

- So far:
  - We defined an API for effects,
  - Used transformers to define types that implement the API.
- We also need a way to execute computations.
- Each transformer is equipped with a function that eliminates it:
  - `runReaderT :: i → ReaderT i m a → m a`
  - `runWriterT :: WriterT i m a → m (a,i)`
  - `runStateT :: i → StateT i m a → m (a,i)`
  - `runExceptionT::ExceptionT i m a → m (Either i a)`

Execution

- So far:
  - We defined an API for effects,
  - Used transformers to define types that implement the API.
- We also need a way to execute computations.
- Each transformer is equipped with a function that eliminates it:
  - `runReaderT :: i → ReaderT i m a → m a`
  - `runWriterT :: WriterT i m a → m (a,i)`
  - `runStateT :: i → StateT i m a → m (a,i)`
  - `runExceptionT::ExceptionT i m a → m (Either i a)`
  - `runContT :: (a → m i) → ContT i m a → m i`

Example

```
test :: IO ()
test  = do x ← runStateT 0
             $ runExceptionT
             $ runReaderT 'q' loop
           print x
  where
  loop = do x ← inBase getChar
            end ← ask
            when (x ≠ end) $
              do when (not (isDigit x)) $ raise "Bad input."
                 let n = fromEnum x - fromEnum '0'
                 tot ← get
                 set (tot + n)
                 loop
```

Orthogonality of Effects

- If the effects are orthogonal, then we can apply the transformers in any order (i.e., they commute).
- Two groups of transformers:
  - Plumbing: `ReaderT`, `WriterT`, `StateT`
  - Control: `ExceptionT`, `ContT`
- Plumbing transformers are orthogonal with each other.
- Control transformers interact with other transformers.

Example

- Interaction of state and exceptions
  - What happens to the state when we raise an exception?
  - Imperative semantics: state is preserved.
  - Transaction semantics: revert changes to the state.

Example

- Interaction of state and exceptions
  - What happens to the state when we raise an exception?
  - Imperative semantics: state is preserved.
  - Transaction semantics: revert changes to the state.
- The order of the transformers determines the behavior.
- "Earlier" transformers take precedence:
  - Imperative: `ExceptionT x (StateT s m)`
  - Transaction: `StateT s (ExceptionT x m)`

Overloaded Execution

- Sometimes it is convenient to "execute" a computation without eliminating the effect
- Think execute in a separate thread w.r.t an effect
- Example—at the outermost level:

```
chCtxt :: (i → i) → ReaderT i m → ReaderT i m
chCtxt f m = do i ← ask
                lift $ runReaderT (f i) m
```

- Next: API that generalizes this example...

RunReader

- Change the context for the duration of a computation.

```
runReaderT    :: i → ReaderT i m a → m a

class (ReaderM m i) ⇒ RunReaderM m i | m ⇝ i where
  local       :: i → m a → m a
```

RunReader

- Change the context for the duration of a computation.

```
runReaderT   :: i → ReaderT i m a → m a

class (ReaderM m i) ⇒ RunReaderM m i | m ⇝ i where
  local      :: i → m a → m a
```

- Example

```
inModCtxt :: (RunReaderM m i) ⇒ (i → i) → m a → m a
inModCtxt f m  = do x ← ask
                    local (f x) m
```

RunWriter

- Collect the output from a sub-computation.

```
runWriterT  :: WriterT i m a → m (a,i)

class (WriterM m i) ⇒ RunWriterM m i | m ⇝ i where
  collect   :: m a → m (a,i)
```

RunWriter

- Collect the output from a sub-computation.

```
runWriterT  :: WriterT i m a → m (a,i)

class (WriterM m i) ⇒ RunWriterM m i | m ⇝ i where
  collect   :: m a → m (a,i)
```

- Example

```
censor :: (RunWriterM m i) ⇒ (i → i) → m a → m a
censor f m = do (a,xs) ← collect m
                put (f xs)
                return a
```

RunException

- Exceptions are explicit in the result.

```
runExceptionT :: ExceptionT i m a → m (Either i a)

class ExceptionM m i ⇒ RunExceptionM m i | m ⇝ i where
  try          :: m a → m (Either i a)
```

RunException

- Exceptions are explicit in the result.

```
runExceptionT :: ExceptionT i m a → m (Either i a)

class ExceptionM m i ⇒ RunExceptionM m i | m ⇝ i where
  try          :: m a → m (Either i a)
```

- Example

```
handle :: (RunExceptionM m x) ⇒ m a → (x → m a) → m a
handle m h  = do r ← try m
                 case r of
                   Left err → h err
                   Right a → return a
```

Summary

- There are three aspects to the library:
  - An (overloaded) API for describing computations,
  - An implementation based on monad transformers,
  - A collection of functions to execute computations.
- Libraries are useful becasue:
  - They save work and increase productivity,
  - Eliminate a potential source of mistakes.

Resources

- Libraries:
  - monadLib (this talk)
    `http://www.galois.com/~diatchki/monadLib`
  - mtl (similar) in GHC's base package
- Some papers:
  - *Monad Transformers and Modular Interpreters*
    Sheng Liang, Paul Hudak, and Mark Jones.
  - *Functional Programming with Overloading and Higher-Order Polymorphism*
    Mark P. Jones