MGS 2006: AFP Lecture 3
*Monad Transformers*

Henrik Nilsson

University of Nottingham, UK
What if we need to support more than one type of effect?

newtype SE s a = SE (s -> Maybe (a, s))
What if we need to support more than one type of effect?

For example: State and Error/Partiality?
What if we need to support more than one type of effect?

For example: State and Error/Partiality?

We could implement a suitable monad from scratch:

```haskell
newtype SE s a = SE (s -> Maybe (a, s))
```
Monad Transformers (2)

However:

- Not always obvious how:
- How to combine state and error and CPS and . . . ?
- Should the combination of state and error have been

\[
\text{newtype SE s a = SE (s -> (Maybe a, s))}
\]

Duplication of effort: similar patterns related to specific effects are going to be repeated over and over in the various combinations.
However:

- Not always obvious how:
Monad Transformers (2)

However:

- Not always obvious how:
  - How to combine state and error and CPS and . . . ?
However:

- Not always obvious how:
  - How to combine state and error and CPS and . . . ?
  - Should the combination of state and error have been

    newtype SE s a = SE (s -> (Maybe a, s))
Monad Transformers (2)

However:

- Not always obvious how:
  - How to combine state and error and CPS and ...?
  - Should the combination of state and error have been

\[
\text{newtype } \text{SE } s \ a = \text{SE } (s \rightarrow (\text{Maybe } a, s))
\]

- Duplication of effort: similar patterns related to specific effects are going to be repeated over and over in the various combinations.
Monad Transformers can help:

*Monad Transformers* can help:
Monad Transformers can help:

- A **monad transformer** transforms a monad by adding support for an additional effect.
Monad Transformers (3)

*Monad Transformers* can help:

- A *monad transformer* transforms a monad by adding support for an additional effect.

- A library of monad transformers can be developed, each adding a specific effect (state, error, ...), allowing the programmer to mix and match.
Monad Transformers can help:

- A _monad transformer_ transforms a monad by adding support for an additional effect.
- A library of monad transformers can be developed, each adding a specific effect (state, error, . . .), allowing the programmer to mix and match.
- A form of _aspect-oriented programming_.

_Monad Transformers_
Lecture 3

- Introduction to Monad Transformers
- Some standard Monad Transformers and their combinations
- A concurrency monad transformer (with an eye to giving semantics too/interpreting a Java-like language)
A monad transformer maps monads to monads. This is represented by a type constructor of the following kind:

\[ T : : (\star \to \star) \to \star \to \star \]
A monad transformer maps monads to monads. This is represented by a type constructor of the following kind:

\[ T :: \quad (\star \to \star) \to \star \to \star \]

Additionally, we require monad transformers to \textit{add} computational effects. Thus we require a mapping from computations in the underlying monad to computations in the transformed monad:

\[ \text{lift} :: M \ a \to T \ M \ a \]
These requirements are captured by the following (multi-parameter) type class:

```haskell
class (Monad m, Monad (t m)) => MonadTransformer t m where
  lift :: m a -> t m a
```
Classes for Specific Effects

A monad transformer adds specific effects to any monad. Thus there can be many monads supporting the same operations. Introduce classes to handle the overloading:

```haskell
class Monad m => E m where
    eFail :: m a
    eHandle :: m a -> m a -> m a
```

```haskell
class Monad m => S m s | m -> s where
    sSet :: s -> m ()
    sGet :: m s
```
The Identity Monad

We are going to construct monads by successive transformations of the identity monad:

```haskell
newtype I a = I a
unI (I a) = a

instance Monad I where
  return a = I a
  m >>= f = f (unI m)

runI :: I a -> a
runI = unI
```
The Error Monad Transformer (1)

newtype ET m a = ET (m (Maybe a))
unET (ET m) = m

instance Monad m => Monad (ET m) where
  return a = ET (return (Just a))

  m >>= f = ET $ do
    ma <- unET m
    case ma of
      Nothing -> return Nothing
      Just a  -> unET (f a)
We need the ability to run transformed monads:

\[
\text{runET} :: \text{Monad } m \Rightarrow \text{ET } m \ a \rightarrow m \ a
\]

\[
\text{runET } \text{etm} = \text{do}
\quad \text{ma } \leftarrow \text{unET } \text{etm}
\quad \text{case } \text{ma} \ of
\quad \quad \text{Just } a \rightarrow \text{return } a
\]

**ET is a monad transformer:**

\[
\text{instance Monad } m \Rightarrow \text{MonadTransformer ET } m \text{ where}
\quad \text{lift } m = \text{ET } (m \gg= \backslash a \rightarrow \text{return } (\text{Just } a))
\]
The Error Monad Transformer (3)

Any monad transformed by ET is an instance of E:

\[
\text{instance Monad m => E (ET m) where}
\]

\[
\text{eFail = ET (return Nothing)}
\]

\[
\text{m1 'eHandle' m2 = ET $ do}
\]

\[
\text{ma <- unET m1}
\]

\[
\text{case ma of}
\]

\[
\text{Nothing -> unET m2}
\]

\[
\text{Just _ -> return ma}
\]
A state monad transformed by \( ET \) is a state monad:

```haskell
instance S m s => S (ET m) s where
  sSet s = lift (sSet s)
  sGet = lift sGet
```
Exercise 1: Running transf. monads

Let

\[
\text{ex1} = \text{eFail} \ 'eHandle' \ \text{return} \ 1
\]

1. Suggest a possible type for \text{ex1}.
2. How can \text{ex1} be run, given your type?
Exercise 1: Solution

```haskell
ex1 :: ET I Int
ex1 = eFail 'eHandle' return 1

ex1r :: Int
ex1r = runI (runET ex1)
```
newtype ST s m a = ST (s -> m (a, s))
unST (ST m) = m

instance Monad m => Monad (ST s m) where
    return a = ST (\s -> return (a, s))

    m >>= f = ST $ \s -> do
        (a, s') <- unST m s
        unST (f a) s'
We need the ability to run transformed monads:

```haskell
code
runST :: Monad m => ST s m a -> s -> m a
runST stf s0 = do
    (a, _) <- unST stf s0
    return a
```

**ST is a monad transformer:**

```haskell
code
instance Monad m => MonadTransformer (ST s) m where
    lift m = ST (\s -> m >>= \a ->
                 return (a, s))
```
The State Monad Transformer (3)

Any monad transformed by \texttt{ST} is an instance of \texttt{S}:

\[
\text{instance Monad } m \Rightarrow S \ (ST \ s \ m) \ s \ where \\
\text{sSet \ } s = ST \ \(_\_ \rightarrow \) return \ ((), \ s)) \\
\text{sGet} = ST \ (_s \rightarrow \) return \ (s, \ s))
\]

An error monad transformed by \texttt{ST} is an error monad:

\[
\text{instance E } m \Rightarrow E \ (ST \ s \ m) \ where \\
\text{eFail} = \text{lift eFail} \\
\text{m1 'eHandle' m2} = ST \ $ \ _s \rightarrow \\
\text{unST m1 s 'eHandle' unST m2 s}
\]
Exercise 2: Effect ordering

Consider the code fragment

```haskell
ex2a :: ST Int (ET I) Int
ex2a= (sSet 3 >> eFail) 'eHandle' sGet
```

Note that the exact same code fragment also can be typed as follows:

```haskell
ex2b :: ET (ST Int I) Int
ex2b = (sSet 42 >> eFail) 'eHandle' sGet
```

What is

```haskell
runI (runET (runST ex2a 0))
runI (runST (runET ex2b) 0)
```
Exercise 2: Solution

\[
\text{runI } (\text{runET } (\text{runST } \text{ex2a } 0)) = 0
\]
\[
\text{runI } (\text{runST } (\text{runET } \text{ex2b}) 0) = 3
\]
Exercise 3: Alternative $ST$?

To think about.

Could $ST$ have been defined in some other way, e.g.

```haskell
newtype ST s m a = ST (m (s -> (a, s)))
```

or perhaps

```haskell
newtype ST s m a = ST (s -> (m a, s))
```
Exercise 4: Continuation monad transf.

The continuation monad transformer is given by:

\[
\text{newtype CPST } r \ m \ a = \text{CPST } ((a \rightarrow m \ r) \rightarrow m \ r)
\]

\[
\text{unCPST} :: \text{CPST } r \ m \ a \rightarrow ((a \rightarrow m \ r) \rightarrow m \ r)\\
\text{unCPST} \ (\text{CPST} \ f) = f
\]

\[
\text{class Monad } m \Rightarrow \text{CPS } m \text{ where}\\
\text{callCC} :: ((a \rightarrow m \ b) \rightarrow m \ a) \rightarrow m \ a
\]

Outline the various instances for CPCT and monads transformed by it.
Exercise 4: Solution (1)

instance Monad m => Monad (CPST r m) where
  return a = undefined
  m >>= f = undefined

instance Monad m => MonadTransformer (CPST r) m where
  lift m = undefined

instance Monad m => CPS (CPST r m) where
  callCC f = undefined
As to effect ordering, making CPST the outer transformer is the natural and easy choice:

```haskell
instance E m => E (CPST r m) where
    eFail = undefined
    m1 'eHandle' m2 = undefined

instance S m s => S (CPST r m) s where
    sSet s = undefined
    sGet = undefined
```
newtype CPST r m a = CPST ((a -> m r) -> m r)

unCPST :: CPST r m a -> ((a -> m r) -> m r)
unCPST (CPST f) = f

instance Monad m => Monad (CPST r m) where
  return a = CPST (\k -> k a)
  m >>= f = CPST $ \k ->
    unCPST m (\a -> unCPST (f a) k)
The Continuation Monad Transformer (2)

We need the ability to run transformed monads:

```haskell
runCPST :: Monad m => CPST a m a -> m a
runCPST m = unCPST m return
```

**CPST is a monad transformer:**

```haskell
instance Monad m => MonadTransformer (CPST r) m where
    lift m = CPST $ \k -> m >>= k
```
The Continuation Monad Transformer (3)

Any monad transformed by $\text{CPST}$ is an instance of $\text{CPS}$:

\[
\text{instance Monad } m \Rightarrow \text{CPS} (\text{CPST } r \ m) \text{ where }
\]
\[
\text{callCC } f = \text{CPST } \text{\$ \ \_ \ -> }
\]
\[
\text{unCPST } (f (\text\_ \ -> \text{CPST } \text{\$ \ \_ \ -> \ k \ a})) \ k
\]
The Continuation Monad Transformer (4)

An error monad transformed by CPST is an error monad:

\[
\text{instance } E \ m \Rightarrow E \ (\text{CPST} \ r \ m) \ \text{where}
\]
\[
e\text{Fail} = \text{lift} \ e\text{Fail}
\]
\[
m_1 \ 'e\text{Handle}' \ m_2 = \text{CPST} \ \backslash k \rightarrow
\]
\[
\text{unCPST} \ m_1 \ k \ 'e\text{Handle}' \ \text{unCPST} \ m_2 \ k
\]

A state monad transformed by CPST is a state monad:

\[
\text{instance } S \ m \ s \Rightarrow S \ (\text{CPST} \ r \ m) \ s \ \text{where}
\]
\[
s\text{Set} \ s = \text{lift} \ (s\text{Set} \ s)
\]
\[
s\text{Get} = \text{lift} \ s\text{Get}
\]
Example: CPS and state (1)

\[ f \::\ (\text{CPS } m, \text{S } m \text{ Int}) \Rightarrow \text{Int} \rightarrow \text{Int} \rightarrow m \text{ (Int,Int)} \]

\[ f \ x \ y = \text{do} \]
\[ \quad \ x \leftarrow \text{callCC} \ \_ \\_ \ \text{\exit} \ \rightarrow \ \text{do} \]
\[ \quad \quad \ \text{let } d = x - y \]
\[ \quad \quad \text{sSet 11} \]
\[ \quad \quad \text{when } (d == 0) \ (\text{exit } (-1)) \]
\[ \quad \quad \text{let } z = (\text{abs } ((x + y) \div d)) \]
\[ \quad \ldots \]
Example: CPS and state (2)

...  
\[ x \leftarrow \text{sGet} \]
\[ \text{sSet} \ (x \times 2) \]
\[ \text{when} \ (z > 10) \ (\text{exit} \ (-2)) \]
\[ x \leftarrow \text{sGet} \]
\[ \text{sSet} \ (x \times 2) \]
\[ \text{return} \ (z^3) \]

\[ s \leftarrow \text{sGet} \]
\[ \text{return} \ (x, s) \]
Example: CPS and state (3)

\[
\text{run } m = \text{runI} \ (\text{runST} \ (\text{runCPST} \ m \ 0))
\]

\[
\text{run } (f \ 10 \ 6) = (64, 44)
\]

\[
\text{run } (f \ 10 \ 10) = (-1, 11)
\]

\[
\text{run } (f \ 10 \ 9) = (-2, 22)
\]
A Concurrency Monad Transformer (1)

class Monad m => GlobalStateMonad m where
    gRead :: m Char
    gWrite :: Char -> m ()
    gPrint :: Char -> m ()

class Monad m => ConcMonad m where
    cFork :: m a -> m ()
    cEnd :: m a
data Thread m = Atom (m (Thread m))
  | Fork (Thread m) (Thread m)
  | End

type ThreadQueue m = [Thread m]

newtype CT m a = CT ((a->Thread m) -> Thread m)

fromCT :: CT m a -> ((a->Thread m) -> Thread m)
fromCT (CT x) = x
A Concurrency Monad Transformer (3)

thread :: Monad m => CT m a -> Thread m
thread m = fromCT m (const End)

instance Monad m => Monad (CT m) where
    return x = CT ($\ k \rightarrow \ k \ x$)
    m >>= f = CT $\ \ k \rightarrow \ fromCT \ m \ (\ x \rightarrow \ fromCT \ (f \ x) \ k)$

instance Monad m => MonadTransformer CT m where
    lift m = CT $\ \ k \rightarrow \ Atom \ (m \gg=\ \ x \rightarrow \ return \ (k \ x))$
instance Monad m => ConcMonad (CT m) where
    cFork m = CT (\k -> Fork (thread m) (k ()))
    cEnd = CT (\_ -> End)
A Concurrency Monad Transformer (5)

runCT :: Monad m => CT m a -> m ()
runCT m = mmap (const ()) (dispatch [] (thread m))

dispatch :: Monad m => ThreadQueue m -> Thread m -> m ()

dispatch rq (Atom m) = m >>= \t ->
                       schedule (rq ++ [t])

dispatch rq (Fork t1 t2) = schedule (rq++[t1,t2])

dispatch rq End = schedule rq

schedule :: Monad m => ThreadQueue m -> m ()

schedule [] = return ()

schedule (t:ts) = dispatch ts t
Example: A concurrent process

```haskell
p3 :: (ConcMonad m, GlobalStateMonad m,
      ErrorMonad m, StateMonad m) => m ()

p3 = do
  gWrite 'Z'
  sWrite 'S'
  cFork p11
  gPrint 'A'
  cFork p2
  gPrint 'B'
  x <- sRead
  gPrint x
  x <- gRead
  gPrint x
```
Reading

