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:

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

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 (3)

\textit{Monad Transformers} can help:
- A \textit{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 \textit{aspect-oriented programming}. 
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)

Monad Transformers in Haskell (1)

- A monad transformer maps monads to monads. This is represented by a type constructor of the following kind:
  \[ T : : (\ast \rightarrow \ast) \rightarrow \ast \rightarrow \ast \]
- Additionally, we require monad transformers to add computational effects. Thus we require a mapping from computations in the underlying monad to computations in the transformed monad:
  \[ \text{lift} :: M a \rightarrow T M a \]

Monad Transformers in Haskell (2)

- These requirements are captured by the following (multi-parameter) type class:
  \[
  \text{class (Monad } m, \text{ Monad } (t \ m)\text{)} \\
  \Rightarrow \text{MonadTransformer } t \ m \text{ where} \\
  \text{lift} :: m a \rightarrow t \ m \ a
  \]

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:

\[
\text{class Monad } m \Rightarrow E \ m \text{ where} \\
\text{eFail} :: m a \\
\text{eHandle} :: m a \rightarrow m a \rightarrow m a
\]

\[
\text{class Monad } m \Rightarrow S \ m \ s \mid m \rightarrow s \text{ where} \\
\text{sSet} :: s \rightarrow m () \\
\text{sGet} :: m s
\]
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)

```haskell
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 _ -> unET (f a)
```

The Error Monad Transformer (2)

We need the ability to run transformed monads:

```haskell
runET :: Monad m => ET m a -> m a
runET etm = do
  ma <- unET etm
  case ma of
    Just a -> return a

ET is a monad transformer:

instance Monad m => MonadTransformer ET m where
  lift m = ET (m >>= \a -> return (Just a))
```

The Error Monad Transformer (3)

Any monad transformed by ET is an instance of E:

```haskell
instance Monad m => E (ET m) where
  eFail = ET (return Nothing)
m1 `eHandle` m2 = ET $ do
  ma <- unET m1
  case ma of
    Nothing -> unET m2
    Just _ -> return ma
```
The Error Monad Transformer (4)

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

```haskell
ex1 = eFail 'eHandle' return 1
```

1. Suggest a possible type for `ex1`.
2. How can `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)
```

The State Monad Transformer (1)

```haskell
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'
```
The State Monad Transformer (2)

We need the ability to run transformed monads:

```haskell
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
instance Monad m => MonadTransformer (ST s) m where
  lift m = ST (
      s -> m >>= \a ->
        return (a, 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)
```

The State Monad Transformer (3)

Any monad transformed by ST is an instance of S:

```haskell
instance Monad m => S (ST s m) s where
  sSet s = ST (\_ -> return ((), s))
  sGet   = ST (\s -> return (s, s))
```

An error monad transformed by ST is an error monad:

```haskell
instance E m => E (ST s m) where
  eFail = lift eFail
  m1 `eHandle` m2 = ST $ \s ->
    unST m1 s `eHandle` unST m2 s
```

Exercise 2: Solution

```haskell
runI (runET (runST ex2a 0)) = 0
runI (runST (runET ex2b) 0) = 3
```
Exercise 3: Alternative ST?

To think about.
Could ST have been defined in some other way, e.g.

\[
\text{newtype } ST \, s \, m \, a = ST \, (m \, (s \rightarrow \, (a, \, s)))
\]
or perhaps

\[
\text{newtype } ST \, s \, m \, a = ST \, (s \rightarrow \, (m \, a, \, s))
\]

Exercise 4: Continuation monad transf.

The continuation monad transformer is given by:

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

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

\[
\text{unCPST} \, (CPST \, f) = f
\]

class Monad m => CPS m 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

Exercise 4: Solution (2)

As to effect ordering, making CPST the outer transformer is the natural and easy choice:

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)

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

CPST is a monad transformer:

instance Monad m => MonadTransformer (CPST r) m where
  lift m = CPST $ \k -> m >>= k

Any monad transformed by CPST is an instance of CPS:

instance Monad m => CPS (CPST r m) where
  callCC f = CPST $ \k ->
    unCPST (f (\a -> CPST $ \_ -> k a)) k

An error monad transformed by CPST is an error monad:

instance E m => E (CPST r m) where
  eFail = lift eFail
  m1 `eHandle` m2 = CPST $ \k ->
    unCPST m1 k `eHandle` unCPST m2 k

A state monad transformed by CPST is a state monad:

instance S m s => S (CPST r m) s where
  sSet s = lift (sSet s)
  sGet = lift sGet
Example: CPS and state (1)

\[
f :: \text{(CPS m, S m Int)} \Rightarrow \text{Int} \rightarrow \text{Int} \rightarrow \text{m (Int,Int)}
f \, x \, y = \text{do}
\begin{align*}
    x &\leftarrow \text{callCC (\lambda \text{exit} \rightarrow \text{do}} \\
    &\left. \text{let d = x} - y \\
    &\left. \text{sSet 11} \\
    &\left. \text{when (d == 0) (exit (-1))} \\
    &\left. \text{let z = (abs ((x + y) \ `div` d))} \\
    &\ldots
\end{align*}
\]

Example: CPS and state (2)

\[
\ldots
\begin{align*}
    x &\leftarrow \text{sGet} \\
    \text{sSet (x * 2)} \\
    \text{when (z > 10) (exit (-2))} \\
    x &\leftarrow \text{sGet} \\
    \text{sSet (x * 2)} \\
    \text{return (z ^ 3)} \\
    s &\leftarrow \text{sGet} \\
    \text{return (x, s)}
\end{align*}
\]

Example: CPS and state (3)

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

\[
\begin{align*}
\text{run (f 10 6)} &\, = \, (64,44) \\
\text{run (f 10 10)} &\, = \, (-1,11) \\
\text{run (f 10 9)} &\, = \, (-2,22)
\end{align*}
\]

A Concurrency Monad Transformer (1)

\[
\text{class Monad m => GlobalStateMonad m where} \\
\text{gRead :: m Char} \\
\text{gWrite :: Char -> m ()} \\
\text{gPrint :: Char -> m ()}
\]

\[
\text{class Monad m => ConcMonad m where} \\
\text{cFork :: m a -> m ()} \\
\text{cEnd :: m a}
\]
A Concurrency Monad Transformer (2)

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 -> k x)

m >>= f = CT $
  \k -> fromCT m (\x -> fromCT (f x) k)

instance Monad m =>
  MonadTransformer CT m where
lift m = CT $
  \k -> Atom (m >>= \x -> return (k x))

A Concurrency Monad Transformer (4)

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 t's 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