MGS 2006: AFP Lecture 3
Monad Transformers

Henrik Nilsson

University of Nottingham, UK

Monad Transformers (1)

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:

```
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
    ```
    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.

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.
- A form of ***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 :: (* -> *) -> * -> *
```

- 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:

```
lift :: M a -> T M a
```


Monad Transformers in Haskell (2)

- These requirements are captured by the following (multi-parameter) type class:

```
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:

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

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:

```
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)
```


The Error Monad Transformer (2)

We need the ability to run transformed monads:

```
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:

```
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:

```
instance S m s => S (ET m) s where
    sSet s = lift (sSet s)
    sGet = lift sGet
```


Exercise 1: Running transf. monads

Let

```
ex1 = eFail ‘eHandle‘ return 1
```

1. Suggest a possible type for `ex1`.
2. How can `ex1` be run, given your type?

Exercise 1: Solution

```
ex1 :: ET I Int
ex1 = eFail ‘eHandle‘ return 1

ex1r :: Int
ex1r = runI (runET ex1)
```


The State Monad Transformer (1)

```
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:

```
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:

```
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 ST is an instance of S:

```
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:

```
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: Effect ordering

Consider the code fragment

```
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:

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

What is

```
runI (runET (runST ex2a 0))
runI (runST (runET ex2b) 0)
```


Exercise 2: Solution

```
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.

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

or perhaps

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


Exercise 4: Continuation monad transf.

The continuation monad transformer is given by:

```
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

class Monad m => CPS m where
    callCC :: ((a -> m b) -> m a) -> 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
```

The Continuation Monad Transformer (1

```
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:

```
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
```


The Continuation Monad Transformer (3

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
```


The Continuation Monad Transformer (4

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 :: (CPS m,S m Int) => Int -> Int -> m (Int,Int)
f x y = do
    x <- callCC $ \exit -> do
             let d = x - y
             sSet 11
             when (d == 0) (exit (-1))
             let z = (abs ((x + y) ‘div‘ d))
             ...
```


Example: CPS and state (2)

```
             ...
             x <- sGet
             sSet (x * 2)
             when (z > 10) (exit (-2))
             x <- sGet
             sSet (x * 2)
             return (zˆ3)
    s <- sGet
    return (x, s)
```


Example: CPS and state (3)

```
run m = runI (runST (runCPST m 0)
```

```
 run (f 10 6)  = (64,44)
run (f 10 10)  = (-1,11)
 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
```

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 ts t
```

Example: A concurrent process

```
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

- Nick Benton, John Hughes, Eugenio Moggi. Monads and Effects. In *International Summer School on Applied Semantics 2000*, Caminha, Portugal, 2000.
- Koen Claessen. A Poor Man's Concurrency Monad. *Journal of Functional Programming*, 9(3), 1999.
- Sheng Liang, Paul Hudak, Mark Jones. Monad Transformers and Modular Interpreters. In *Proceedings of the 22nd ACM Symposium on Principles of Programming Languages (POPL'95)*, January 1995, San Francisco, California