Code (Wednesday Week 6)

Tuple-based Applicative Formulation

Haskell

```haskell
-- class  Functor f  where
--    fmap        :: (a -> b) -> f a -> f b
-- class Functor f => Applicative f where
--    pure :: a -> f a
--   (<*>) :: f (a -> b) -> f a -> f b

-- It's possible to express Applicative equivalently using this
-- operation as primitive:
class Functor f => App f where
  pure :: a -> f a
  tuple :: f a -> f b -> f (a,b)

-- Using this operation, fmap and pure, implement <*>.
(<**>) :: App f => f (a -> b) -> f a -> f b
(<**>) fab fa = fmap (\(f,x) -> f x) (tuple fab fa)

-- And, using <*> and pure, implement tuple.
-- fmap :: (a -> b) -> f a -> f b
-- fmap ::  (a -> (b -> (a,b))) -> fa -> f (b -> (a,b))
tuple' :: Applicative f => f a -> f b -> f (a,b)
tuple' fa fb = (fmap (,) fa) <*> fb
tuple'' fa fb = (,) <$> fa <*> fb
```

Join-based Monad Formulation

```haskell
-- It's also possible to express Monad using this alternative operation:
class Applicative m => Mon m where
  join :: m (m a) -> m a

(>>>=) :: Mon m => m a -> (a -> m b) -> m b
(>>>=) a f = join (fmap f a)

join' :: Monad m => m (m a) -> m a
join' a =  a >>= id
```

Binary tree applicative functor

```
data Tree a
   = Leaf
   | Node a (Tree a) (Tree a)
    deriving (Show)

instance Functor Tree where
    fmap f Leaf = Leaf
    fmap f (Node x t1 t2)
        = Node (f x) (fmap f t1) (fmap f t2)

instance Applicative Tree where
    pure x  = Node x (pure x) (pure x) -- infinite tree
    Node f fl fr <*> Node x xl xr = Node (f x) (fl <*> xl) (fr <*> xr)
    Leaf <*> _ = Leaf
    _ <*> Leaf = Leaf
```

Formula monad example

```
import Control.Monad (ap)

data Variable = A | B | C deriving (Show,Eq)

data Formula v = Var v
               | Plus (Formula v) (Formula v)
               | Times (Formula v) (Formula v)
               | Constant Int
               deriving (Eq,Show)
example :: Formula Variable
example = Plus (Times (Var A) (Var A)) (Times (Var B) (Var C))

instance Functor Formula where -- fmap is renaming variables
   -- fmap :: (a->b) -> Formula a -> Formula b
   fmap f (Var x)      = Var (f x)
   fmap f (Plus a b)   = Plus (fmap f a) (fmap f b)
   fmap f (Times a b)  = Times (fmap f a) (fmap f b)
   fmap f (Constant i) = Constant i

 -- try fmap (const B) example

 -- Applicatives don't make much sense here, so we can use
 -- the `ap` function from Control.Monad to implement <*>
 -- in terms of >>=, which is easier in this case to write than <*>:
instance Applicative Formula where
 --  (<*>) :: Formula (a -> b) -> Formula a -> Formula b
```

```
   (<*>) = ap
    where
      ap :: Monad m => m (a -> b) -> m a -> m b
      ap f a = do
                 f' <- f
                 a' <- a
                 pure (f' a')
--   pure :: a -> Formula a
   pure = Var

instance Monad Formula where  -- >>= is substitution
   -- (>>=) :: Formula a -> (a -> Formula b) -> Formula b
   Var x      >>= f = f x
   Plus a b   >>= f = Plus (a >>= f) (b >>= f)
   Times a b  >>= f = Times (a >>= f) (b >>= f)
   Constant x >>= f = Constant x

subst A = Constant 3
subst B = Constant 6
subst C = Constant 1

-- try `example >>= subst`
```