Advanced Functional Programming

Tim Sheard
Portland State University

Lecture 2: More about Type Classes

- Implementing Type Classes
- Higher Order Types
- Multi-parameter Type Classes

Implementing Type Classes

- I know of two methods for implementing type classes
- Using the “Dictionary Passing Transform”
- Passing runtime representation of type information.

Source & 2 strategies

class Equal a where
  equal :: a -> a -> Bool
 	
class Nat a where
  inc :: a -> a
  dec :: a -> a
  zero :: a -> Bool
	data EqualL a = EqualL
	  { equalM :: a -> a -> Bool
	  }
	
	data NatL a = NatL
	  { incM :: a -> a
	  , decM :: a -> a
	  , zeroM :: a -> Bool
	  }
		equalX :: Rep a -> a -> a -> Bool
		
		incX :: Rep a -> a -> a
		decX :: Rep a -> a -> a
		zeroX :: Rep a -> a -> Bool

f0 :: (Equal a, Nat a) =>
      a -> a
f0 x =
  if zero x
    && equal x x
  then inc x
  else dec x
	f1 :: EqualL a -> NatL a ->
	      a -> a
	f1 el nl x =
	  if zeroM nl x
	    && equalM el x x
	  then incM nl x
	  else decM nl x
		f2 :: Rep a ->
		      a -> a
		f2 r x =
		  if zeroX r x
		    && equalX r x x
		  then incX r x
		  else decX r x

“Dictionary passing” instances

```
instance Equal Int where


 equal x y = x==y
```

```
instance_l1 :: EqualL Int
instance_l1 =
   EqualL {equalM = equal }  where
  equal x y = x==y
```

```
instance Nat Int where




  inc x = x+1
  dec x = x+1
  zero 0 = True
  zero n = False
```

```
instance_l2 :: NatL Int
instance_l2 =
  NatL {incM=inc,decM=dec,zeroM=zero}
       where
  inc x = x+1
  dec x = x+1
  zero 0 = True
  zero n = False
```

Instance declarations

```
data N = Z | S N

instance Equal N where
  equal Z Z = True
  equal (S x) (S y) = equal x y
  equal _ _ = False

instance Nat N where
  inc x = S x
  dec (S x) = x
  zero Z = True
  zero (S _) = False
```

Become record definitions

```
instance_l3 :: EqualL N
instance_l3 = EqualL { equalM = equal } where
  equal Z Z = True
  equal (S x) (S y) = equal x y
  equal _ _ = False

instance_l4 :: NatL N
instance_l4 =
  NatL {incM = inc, decM = dec, zeroM = zero } where
  inc x = S x
  dec (S x) = x
  zero Z = True
  zero (S _) = False
```

Dependent classes

```
instance Equal a => Equal [a] where
  equal [] [] = True
  equal (x:xs) (y:ys) = equal x y && equal xs ys
  equal _ _ = False

instance Nat a => Nat [a] where
  inc xs = map inc xs
  dec xs = map dec xs
  zero xs = all zero xs
```

become functions between records

```
instance_l5 :: EqualL a -> EqualL [a]
instance_l5 lib = EqualL { equalM = equal } where
  equal [] [] = True
  equal (x:xs) (y:ys) = equalM lib x y && equal xs ys
  equal _ _ = False

instance_l6 :: NatL a -> NatL [a]
instance_l6 lib = NatL { incM = inc, decM =dec, zeroM = zero } where
  inc xs = map (incM lib) xs
  dec xs = map (decM lib) xs
  zero xs = all (zeroM lib) xs
```

In run-time type passing

Collect all the instances together to make one function which has an extra arg which is the representation of the type this function is specialized on.

```
incX (Int p)    x = to p (inc (from p x)) where inc x = x+1
incX (N p)      x = to p (inc (from p x)) where inc x = S x
incX (List a p) x = to p (inc (from p x)) where inc xs = map (incX a) xs


decX (Int p)    x = to p (dec (from p x)) where dec x = x+1
decX (N p)      x = to p (dec (from p x)) where dec x = S x
decX (List a p) x = to p (dec (from p x)) where dec xs = map (decX a) xs



zeroX (Int p)    x = zero (from p x) where zero 0 = True
                                           zero n = False
zeroX (N p)      x = zero (from p x) where zero Z = True
                                           zero (S _) = False
zeroX (List a p) x = zero (from p x) where zero xs = all (zeroX a) xs
```

```
data Proof a b = Ep{from :: a->b, to:: b->a}

data Rep t
  =                 Int  (Proof t Int)
  |                 Char (Proof t Char)
  |                 Unit (Proof t ())
  | forall a b . Arr  (Rep a) (Rep b) (Proof t (a->b))
  | forall a b . Prod (Rep a) (Rep b) (Proof t (a,b))
  | forall a b . Sum  (Rep a) (Rep b) (Proof t (Either a b))
  |                 N    (Proof t N)
  | forall a   . List (Rep a) (Proof t [a])
```

Note how recursive calls at different types are calls to the run-time passing versions with new type-rep arguments.

```
equalX (Int p)    x y = h equal p x y where equal x y = x==y
equalX (N p)      x y = h equal p x y where equal Z Z = True
                                            equal (S x) (S y) = equal x y
                                            equal _ _ = False
equalX (List a p) x y = h equal p x y where equal [] [] = True
                                            equal (x:xs) (y:ys) =
                                               equalX a x y && equal xs ys
                                            equal _ _ = False

h equal p x y = equal (from p x) (from p y)
```

Higher Order types

Type constructors are higher order since they take types as input and return types as output.

Some type constructors (and also some class definitions) are even higher order, since they take type constructors as arguments.

Haskell's Kind system

A Kind is haskell's way of "typing" types

Ordinary types have kind *

```
Int :: *
[ String ] :: *
```

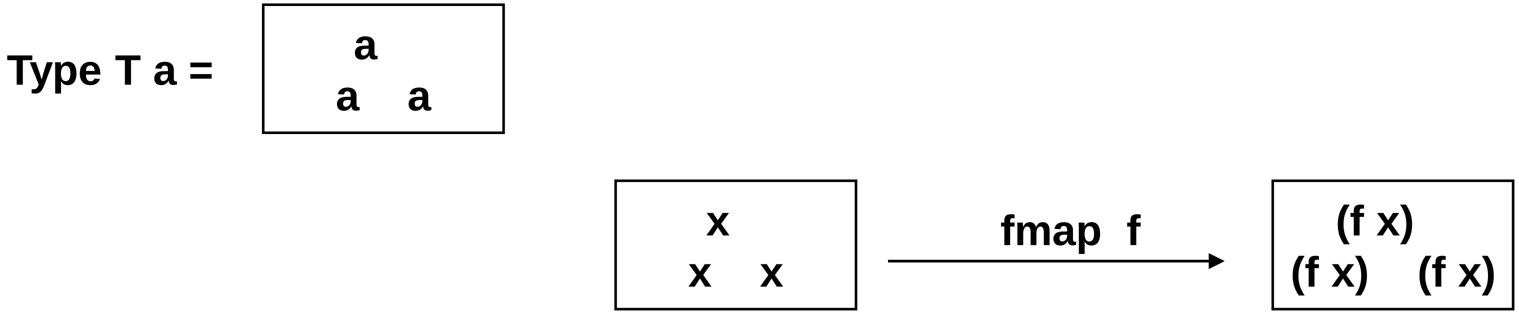
Type constructors have kind `* -> *`

```
Tree :: * -> *
[] :: * -> *
(,) :: * -> * -> *
```

The Functor Class

```
class Functor f where
    fmap :: (a -> b) -> (f a -> f b)
```

- Note how the **`class Functor`** requires a type constructor of kind **`* -> *`** as an argument.
- The method **`fmap`** abstracts the operation of applying a function on every parametric Argument.



More than just types

Laws for **`Functor.`** Most class definitions have some implicit laws that all instances should obey. The laws for Functor are:

```
fmap id = id

fmap (f . g) = fmap f . fmap g
```

Built in Higher Order Types

Special syntax for built in type constructors

```
(->) :: * -> * -> *
[] :: * -> *
(,) :: * -> * -> *
(,,) :: * -> * -> * -> *

type Arrow = (->) Int Int
type List = [] Int
type Pair = (,) Int Int
type Triple = (,,) Int Int Int
```

Instances of class functor

```
data Tree a = Leaf a
            | Branch (Tree a) (Tree a)

instance Functor Tree where
  fmap f (Leaf x) = Leaf (f x)
  fmap f (Branch x y) =
        Branch (fmap f x) (fmap f y)


instance Functor ((,) c) where
   fmap f (x,y) = (x, f y)
```

More Instances

```
instance Functor [] where
    fmap f []     = []
    fmap f (x:xs) = f x : fmap f xs


instance Functor Maybe where
    fmap f Nothing  = Nothing
    fmap f (Just x) = Just (f x)
```

Other uses of Higher order T.C.'s

```
data Tree t a = Tip a
              | Node (t (Tree t a))

t1 = Node [Tip 3, Tip 0]
   Main> :t t1
   t1 :: Tree [] Int

data Bin x = Two x x

t2 = Node (Two(Tip 5) (Tip 21))
   Main> :t t2
   t2 :: Tree Bin Int
```

What is the kind of Tree?

- Tree is a binary type constructor
  - It's kind will be something like:

```
? -> ? -> *
```

- The first argument to Tree is itself a type constructor, the second is just an ordinary type.

```
Tree :: ( * -> *) -> * -> *
```

Another Higher Order Class

Note m is a
type constructor

```
class Monad m where
    (>>=)  :: m a -> (a -> m b) -> m b
    (>>)   :: m a -> m b -> m b
    return :: a -> m a
    fail   :: String -> m a

    p >> q  = p >>= \ _ -> q
    fail s  = error s
```

We pronounce `>>=` as “bind”

and `>>` as “sequence”

Default methods

Note that Monad has two default definitions

```
p >> q   = p >>= \ _ ->
 q
fail s   = error s
```

These are the definitions that are usually correct, so when making an instance of class Monad, only two defintions (`>>=`) and (return) are usually given.

Do notation shorthand

The Do notation is shorthand for the infix operator >>=

```
do e  => e

do { e1 ; e2; … ; en}  =>
     e1  >>   do { e2 ; … ;en}

do { x <- e; f}  =>    e >>= (\ x -> f)
```

where x is a variable

```
do { pat <- e1 ; e2 ; … ; en }  =>
   let ok pat = do { e2; … ; en }
       ok _ = fail “some error message”
   in e1  >>=  ok
```

Monad's and Actions

- We've always used the do notation to indicate an impure computation that performs an actions and then returns a value.
- We can use monads to "invent" our own kinds of actions.
- To define a new monad we need to supply a monad instance declaration.

Example: The action is potential failure

```
instance Monad Maybe where
     Just x  >>= k  =  k x
     Nothing >>= k  =  Nothing
     return         =  Just
```

Example

```
find :: Eq a => a -> [(a,b)] -> Maybe b
find x [] = Nothing
find x ((y,a):ys) =
    if x == y then Just a else find x ys

test a c x =
  do { b <- find a x; return (c+b) }
```

What is the type of test?

What does it return if the find fails?

Multi-parameter Type Classes

- A relationship between two types

```
class (Monad m,Same ref) =>
  Mutable ref m where
   put :: ref a -> a -> m ()
   get :: ref a -> m a
   new :: a -> m (ref a)

class Same ref where
   same :: ref a -> ref a -> Bool
```

An Instance

```
instance
  Mutable (STRef a) (ST a) where
    put = writeSTRef
    get = readSTRef
    new = newSTRef

instance Same (STRef a) where
  same x y = x==y
```

Another Instance

```
instance Mutable IORef IO where
  new = newIORef
  get = readIORef
  put = writeIORef

instance Same IORef where
  same x y = x==y
```

Another Multi-parameter Type Class

```
class Name term name where
  isName :: term -> Maybe name
  fromName :: name -> term

type Var = String
data Term0 =
   Add0 Term0 Term0
 | Const0 Int
 | Lambda0 Var Term0
 | App0 Term0 Term0
 | Var0 Var

instance Name Term0 Var where
  isName (Var0 s) = Just s
  isName _ = Nothing
  fromName s = Var0 s
```

Yet Another

```
class Mult a b c where
  times :: a -> b -> c

instance Mult Int Int Int where
  times x y  = x * y

instance Ix a =>
 Mult Int (Array a Int) (Array a Int)
    where
      times x y = fmap (*x) y
```

An Example Use

- Unification of types is used for type inference.

```
data Type ref m  where
       Tvar :: (Mutable ref m ) =>
               ref (Maybe (Type ref m)) -> Type ref m
       Tgen:: Int -> Type ref m
       Tarrow::Type ref m -> Type ref m -> Type ref m
       Ttuple::  [Type ref m] -> Type ref m
       Tcon:: String -> [Type ref m] -> Type ref m
```

Questions

What are the types of the constructors

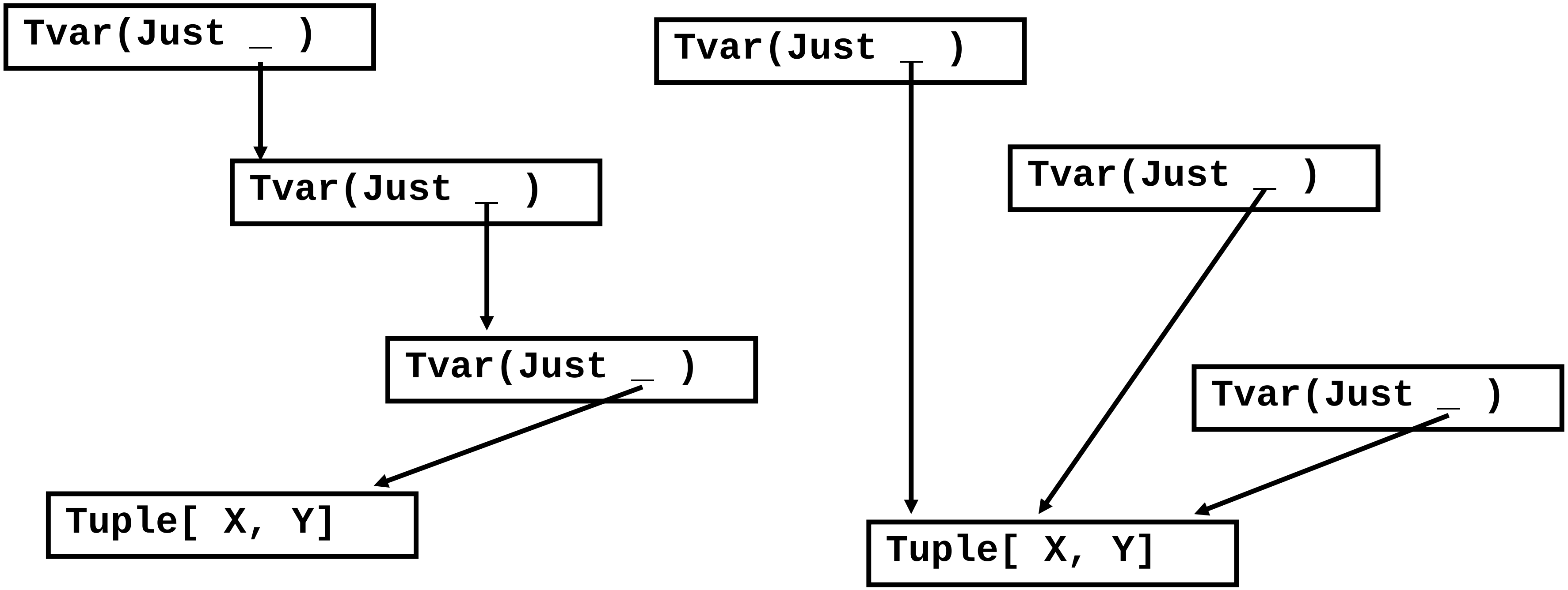
```
Tvar ::

Tgen ::

Tarrow ::
```

Useful Function

Run down a chain of Type TVar references making them all point to the last item in the chain.



Prune

```
prune :: (Monad m, Mutable ref m) =>
          Type ref m -> m (Type ref m)

prune (typ @ (Tvar ref)) =
   do { m <- get ref
      ; case m of
          Just t -> do { newt <- prune t
                       ; put ref (Just newt)
                       ; return newt
                       }
          Nothing -> return typ}
prune x = return x
```

Does a reference occur in a type?

```
occursIn :: Mutable ref m =>
      ref (Maybe (Type ref m)) -> Type ref m -> m Bool
occursIn ref1 t =
 do { t2 <- prune t
    ; case t2 of
        Tvar ref2 -> return (same ref1 ref2)
        Tgen n -> return False
        Tarrow a b ->
           do { x <- occursIn ref1 a
              ; if x then return True
                     else occursIn ref1 b }
        Ttuple xs ->
           do { bs <- sequence(map (occursIn ref1) xs)
              ; return(any id bs)}
        Tcon c xs ->
           do { bs <- sequence(map (occursIn ref1) xs)
              ; return(any id bs) }
    }
```

Unify

```
unify :: Mutable ref m =>
  (Type ref m -> Type ref m -> m [String]) ->
          Type ref m -> Type ref m -> m [String]
unify occursAction x y =
  do { t1 <- prune x
     ; t2 <- prune y
     ; case (t1,t2) of
        (Tvar r1,Tvar r2) ->
           if same r1 r2
              then return []
              else do { put r1 (Just t2); return []}
        (Tvar r1,_) ->
           do { b <- occursIn r1 t2
              ; if b then occursAction t1 t2
                     else do { put r1 (Just t2)
                             ; return [] }
              }
```

Unify continued

```
unify occursAction x y =
  do { t1 <- prune x
     ; t2 <- prune y
     ; case (t1,t2) of
        . . .
        (_,Tvar r2) -> unify occursAction t2 t1
        (Tgen n,Tgen m) ->
            if n==m then return []
                    else return ["generic error"]
        (Tarrow a b,Tarrow x y) ->
          do { e1 <- unify occursAction a x
             ; e2 <- unify occursAction b y
             ; return (e1 ++ e2)
             }
        (_,_) -> return ["shape match error"]
    }
```

Generic Monad Functions

```
sequence  :: Monad m => [m a] -> m [a]
sequence =  foldr mcons (return [])
  where mcons p q =
      do { x <- p
         ; xs <- q
         ; return (x:xs)
         }


mapM :: Monad m => (a -> m b) -> [a] -> m [b]
mapM f as  =  sequence (map f as)
```