Object encoding in Haskell

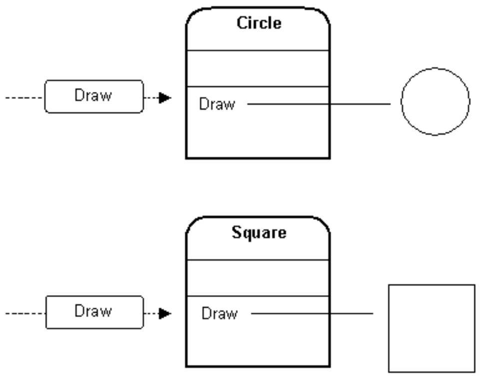
Ralf Lämmel
Universität Koblenz-Landau, Software Languages Team, Koblenz, Germany

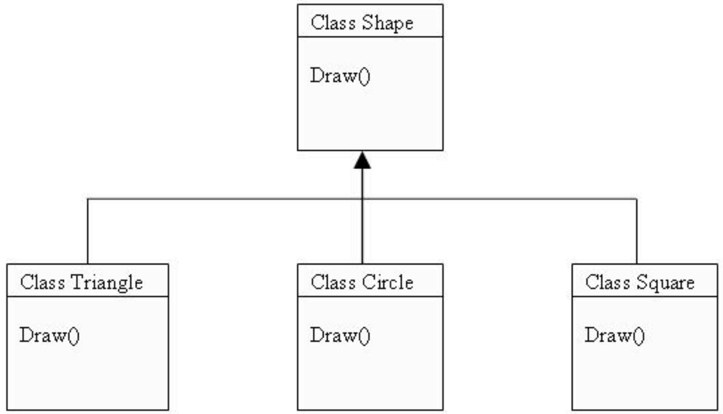
joint work with

Oleg Kiselyov
Fleet Numerical Meteorology and Oceanography Center, Monterey, CA

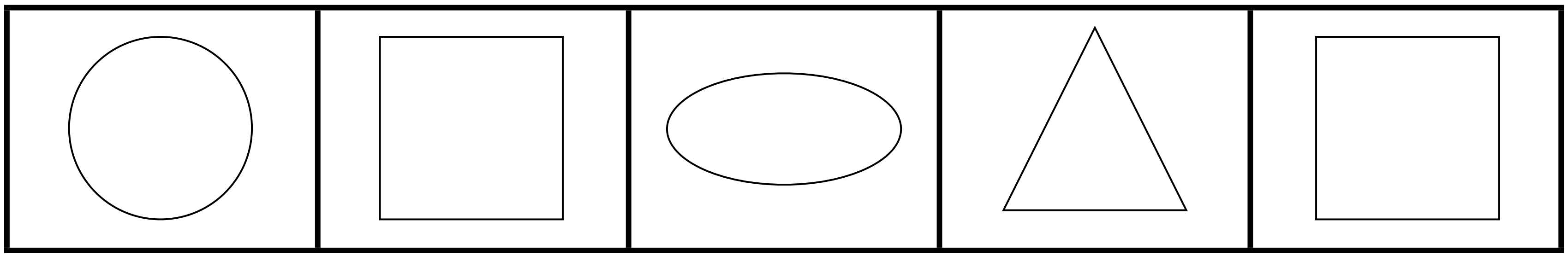
Object encoding in Haskell

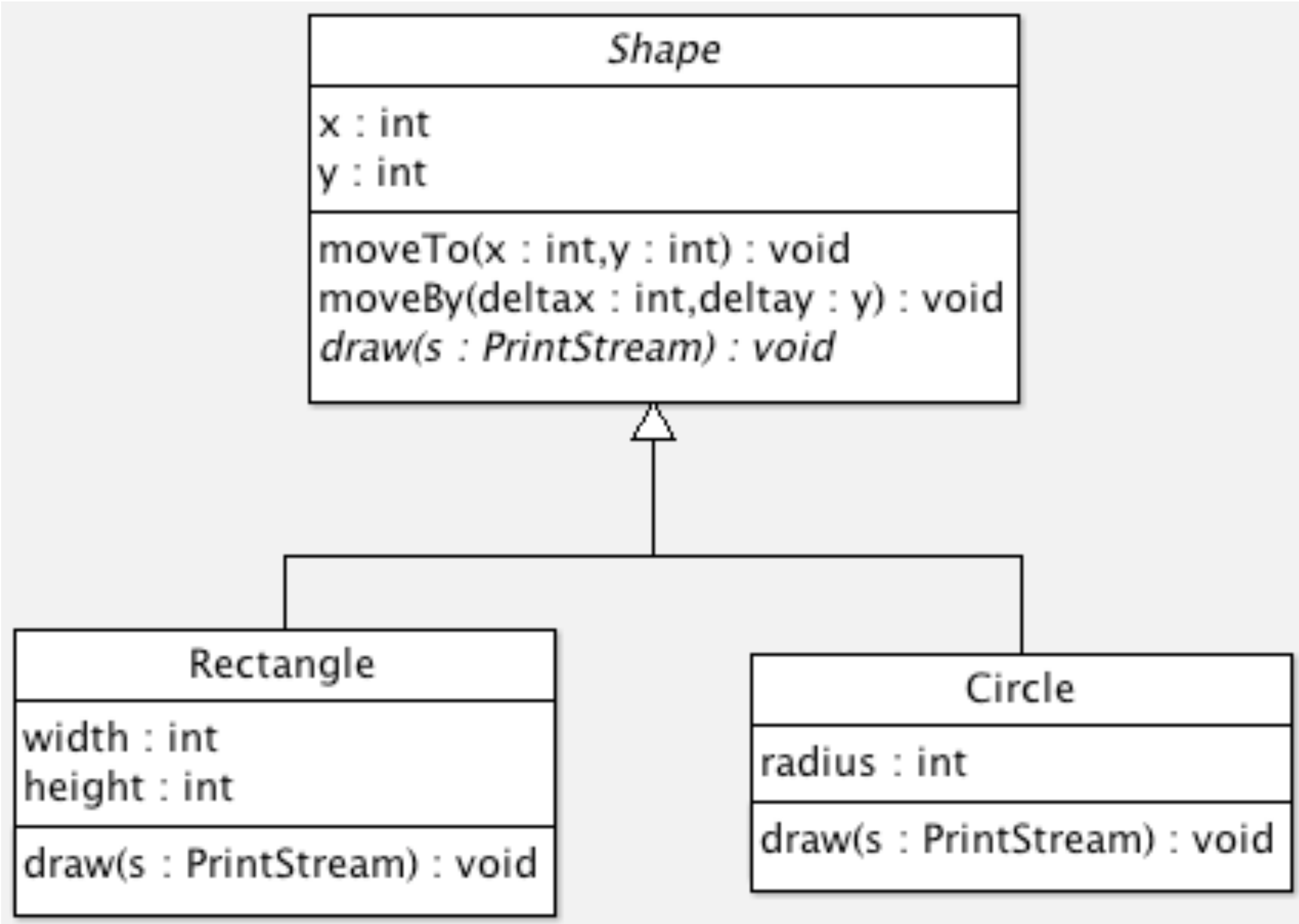
- What's it?
  - Represent the OO tenet in the FP language Haskell
  - Explore different options of representing inheritance, etc.
- Why do I need it?
  - Understand OO more deeply by "encoding".
  - Understand FP more deeply by this challenge.





A benchmark for OO: The Shapes Problem




Shape
x : int
y : int
moveTo(x : int,y : int) : void
moveBy(deltax : int,deltay : y) : void
draw(s : PrintStream) : void
Rectangle
width : int
height : int
draw(s : PrintStream) : void
Circle
radius : int
draw(s : PrintStream) : void

```java
// The abstract base class of all kinds of shapes
public abstract class Shape {

   // Private state
    private int x;
    private int y;

    // Construction is invoked by concrete subclasses
    protected Shape(int x, int y) { moveTo(x, y); }

    // Getters and setters for coordinates
    public int getX() { return x; }
    public int getY() { return y; }
    public void setX(int x) { this.x = x; }
    public void setY(int y) { this.y = y; }

    // Move the shape, absolutely
    public void moveTo(int x, int y) { setX(x); setY(y); }

    // Move the shape, relatively
    public void moveBy(int deltax, int deltay) {
        moveTo(getX() + deltax, getY() + deltay);
    }

    // Draw the shape
    public abstract void draw(PrintStream s);
}
```

```java
// Circle as a kind of shape
class Circle extends Shape {

   // Private state
   private int radius;

   // Constructor
   public Circle(int x, int y, int radius) {
      super(x, y);
      setRadius(radius);
   }

   // Getter and setter for the radius
   public int getRadius() { return radius; }
   public void setRadius(int radius) { this.radius = radius; }

   // Draw the circle
   public void draw(PrintStream s) {
      s.println(
         "Drawing a Circle at:("
         + getX() + ", " + getY()
         + "), radius " + getRadius());
   }
}
```

```java
public static void main(String[] args) {

        // Construct a list of shapes
        Shape scribble[] = new Shape[2];
        scribble[0] = new Rectangle(10, 20, 5, 6);
        scribble[1] = new Circle(15, 25, 8);

        // Handle the shapes in the list polymorphically
        for (int i = 0; i < scribble.length; i++) {
           scribble[i].draw(System.out);
           scribble[i].moveBy(100, 100);
           scribble[i].draw(System.out);
        }
}
```

Object encoding in Haskell

- **Non-encapsulating encoding**
  - Separation of state and methods
  - Methods as regular/overloaded functions
  - Options for state:
    - Closed data type
    - Tail-polymorphic record type
      - Refined options
        - Fixed union type for tail
        - Existential quantification for tail

Many options omitted!
(mainly: Composition and IOP)

- **Encapsulating encoding**
  - Objects as records of state and methods
  - ***Functional objects***
    - Copy semantics for mutation
    - Refined options
      - Fixed union type for tail
      - Existential quantification for tail
      - Narrowing operation for tail
  - ***Mutable objects***
    - Refined options
      - Existential quantification for tail
      - Narrowing operation for tail
      - Use of heterogenous lists

Object encoding in Haskell: Non-encapsulating; Closed data type

```haskell
-- The datatype for all forms of shapes

data Shape =
           Rectangle { getX      :: Int
                     , getY      :: Int
                     , getWidth  :: Int
                     , getHeight :: Int
                     }
         |
           Circle { getX      :: Int
                  , getY      :: Int
                  , getRadius :: Int
                  }
```

```haskell
-- Total setters

setX :: Int -> Shape -> Shape
setX i s = s { getX = i }

setY :: Int -> Shape -> Shape
setY i s = s { getY = i }


-- Partial setters

setWidth :: Int -> Shape -> Shape
setWidth i s = s { getWidth = i }

setHeight :: Int -> Shape -> Shape
setHeight i s = s { getHeight = i }

setRadius :: Int -> Shape -> Shape
setRadius i s = s { getRadius = i }
```

```haskell
-- Moving shapes

moveTo :: Int -> Int -> Shape -> Shape
moveTo x y = setY y . setX x

moveBy :: Int -> Int -> Shape -> Shape
moveBy deltax deltay s = moveTo x y s
 where x = getX s + deltax
       y = getY s + deltay
```

```haskell
-- A function for drawing shapes

draw :: Shape -> IO ()

draw s@(Rectangle _ _ _ _)
    =  putStrLn $ concat ["Drawing a Rectangle at:",
                          show (getX s,getY s),
                          ", width ",  show (getWidth s),
                          ", height ", show (getHeight s)]

draw s@(Circle _ _ _)
    =  putStrLn $ concat ["Drawing a Circle at:",
                          show (getX s,getY s),
                          ", radius ", show (getRadius s)]
```

```haskell
-- Test case for heterogeneous collections

main = do

         -- Construct a list of shapes
         let scribble = [ Rectangle 10 20 5 6
                        , Circle 15 25 8
                        ]

         -- Handle the shapes in the list polymorphically
         mapM_ (\s -> do draw s
                         draw (moveBy 100 100 s))
               scribble
```

Object encoding in Haskell: Non-encapsulating; Tail-polymorphic record type

```haskell
-- Tail-polymorphic shapes

data Shape w =
     Shape { getX :: Int
           , getY :: Int
           , shapeTail :: w }


-- The constructor for shapes

shape x y tail = Shape { getX = x
                       , getY = y
                       , shapeTail = tail }
```

```haskell
-- Non-overridable functionality on shapes

setX, setY :: Int -> Shape w -> Shape w
setX i s = s { getX = i }
setY i s = s { getY = i }

moveTo, moveBy :: Int -> Int -> Shape w -> Shape w
moveTo x y = setY y . setX x
moveBy deltax deltay s = moveTo x y s
  where x = getX s + deltax
        y = getY s + deltay


-- A class for a type-specific drawing method

class Draw w where
  draw :: Shape w -> IO ()
```

```haskell
-- Tail-polymorphic tails of circles

data CircleDelta w =
     CircleDelta { getRadius  :: Int
                 , circleTail :: w }


-- Circles as tail-instantiated shapes

type Circle w = Shape (CircleDelta w)


-- The constructor for circles

circle :: Int -> Int -> Int -> Circle ()
circle x y radius
 = shape x y $ CircleDelta { getRadius  = radius
                           , circleTail = () }
```

```haskell
-- Circle-specific setter for radius

setRadius :: Int -> Circle w -> Circle w
setRadius i s = s { shapeTail = (shapeTail s) { getRadius = i } }


-- Circle-specific implementation of draw method

instance Draw (CircleDelta w) where
  draw s
    =  putStrLn $ concat ["Drawing a Circle at:",
                          show (getX s,getY s),
                          ", radius ",
                          show (getRadius (shapeTail s))]
```

Type error!

```haskell
-- Test case for heterogeneous collections

main = do

        -- Construct a list of shapes
        let scribble = [ rectangle 10 20 5 6
                       , circle 15 25 8
                       ]

        -- Handle the shapes in the list polymorphically
        mapM_ (\s ->  do draw s
                         draw (moveBy 100 100 s))
              scribble
```

Tail-polymorphic record type *with a fixed union type for tail*

```haskell
-- Test case for heterogeneous collections

main = do

        -- Construct a list of shapes
        let scribble = [ upCastToShape (rectangle 10 20 5 6)
                       , upCastToShape (circle 15 25 8)
                       ]

        -- Handle the shapes in the list polymorphically
        mapM_ (\s ->  do draw s
                         draw (moveBy 100 100 s))
              scribble
```

```haskell
-- Define a closed union over two kinds of shapes

type AnyShape w = Shape (Either (RectangleDelta w) (CircleDelta w))


-- Define embedding into union as on overloaded operation

class UpCastToShape d where
  upCastToShape :: Shape (d w) -> AnyShape w

instance UpCastToShape RectangleDelta where
  upCastToShape = tagShape Left

instance UpCastToShape CircleDelta where
  upCastToShape = tagShape Right


-- Tag the shape delta as needed for embedding into Either

tagShape :: (w -> w') -> Shape w -> Shape w'
tagShape f s = s { shapeTail = f (shapeTail s) }
```

```haskell
-- Define draw for tagged shapes

instance (Draw a, Draw b) => Draw (Either a b) where
  draw = eitherShape draw draw


-- Discriminate on Either-typed tail of shape

eitherShape :: (Shape w -> t) -> (Shape w' -> t) -> Shape (Either w w') -> t
eitherShape f g s
  = case shapeTail s of
      (Left s')  -> f (s { shapeTail = s' })
      (Right s') -> g (s { shapeTail = s' })
```

Tail-polymorphic record type *with existential quantification for tail*

```haskell
-- Existential envelope for `drawables'

data AnyShape = forall a. Draw a
  => AnyShape (Shape a)

draw' (AnyShape s) = draw s
moveTo' x y (AnyShape s) = AnyShape $ moveTo x y s
moveBy' dx dy (AnyShape s) = AnyShape $ moveBy dx dy s


-- Test case for heterogeneous collections

main =
      do
         -- Construct a list of shapes
         let scribble = [ AnyShape (rectangle 10 20 5 6)
                        , AnyShape (circle 15 25 8)
                        ]

         -- Handle the shapes in the list polymorphically
         mapM_ (\s -> do draw' s
                         draw' (moveBy' 100 100 s))
               scribble
```

Object encoding in Haskell: **Functional objects**

```haskell
-- Recursive type for functional shape objects

data Shape w =
     Shape { getX      :: Int
           , getY      :: Int
           , setX      :: Int -> Shape w
           , setY      :: Int -> Shape w
           , moveTo    :: Int -> Int -> Shape w
           , moveBy    :: Int -> Int -> Shape w
           , draw      :: IO ()
           , shapeTail :: w
           }


-- The constructor for shapes

shape x y draw tail
  = Shape { getX      = x
          , getY      = y
          , setX      = \x -> shape x y draw tail
          , setY      = \y -> shape x y draw tail
          , moveTo    = \x y -> shape x y draw tail
          , moveBy    = \dx dy -> shape (x+dx) (y+dy) draw tail
          , draw      = draw x y
          , shapeTail = tail
          }
```

```haskell
-- Tail-polymorphic tails of circles

data CircleDelta w =
     CircleDelta { getRadius'  :: Int
                 , setRadius'  :: Int -> Circle w
                 , circleTail  :: w
                 }


--- Circles as tail-instantiated shapes

type Circle w = Shape (CircleDelta w)


-- Convenient access to nested parts

getRadius = getRadius' . shapeTail
setRadius = setRadius' . shapeTail


-- The constructor for circles

circle x y radius = shape x y draw tail
  where
    draw x y
        =  putStrLn $ concat ["Drawing a Circle at:",
                              show (x,y),
                              ", radius ",
                              show radius]

    tail = CircleDelta { getRadius' = radius
                       , setRadius' = \radius -> circle x y radius
                       , circleTail = () }
```

Functional objects *with a fixed union type for tail*

```haskell
-- Test case for heterogeneous collections

main = do

         -- Construct a list of shapes
         let scribble = [ upCastToShape (rectangle 10 20 5 6)
                        , upCastToShape (circle 15 25 8)
                        ]

         -- Handle the shapes in the list polymorphically
         mapM_ (\s -> do draw s
                         draw (moveBy s 100 100))
               scribble
```

```haskell
-- Define a closed union over two kinds of shapes

type AnyShape w = Shape (Either (RectangleDelta w) (CircleDelta w))


-- Define embedding into union as on overloaded operation

class UpCastToShape d
 where
  upCastToShape :: Shape (d w) -> AnyShape w

instance UpCastToShape RectangleDelta
 where
  upCastToShape = tagShape Left

instance UpCastToShape CircleDelta
 where
  upCastToShape = tagShape Right


-- Tag the shape delta as needed for embedding into Either

tagShape :: (w -> w') -> Shape w -> Shape w'
tagShape f s = s { setX      = tagShape f . setX s
                 , setY      = tagShape f . setY s
                 , moveTo    = \z -> tagShape f . moveTo s z
                 , moveBy    = \z -> tagShape f . moveBy s z
                 , shapeTail = f (shapeTail s) }
```

Functional objects *with existential quantification for tail*

```haskell
-- Existential envelope for shapes

data AnyShape = forall x.
     AnyShape (Shape x)

draw'   (AnyShape s) = draw s
moveTo' (AnyShape s) x y = AnyShape $ moveTo s x y
moveBy' (AnyShape s) dx dy = AnyShape $ moveBy s dx dy


-- Test case for heterogeneous collections

main =
      do

         -- Construct a list of shapes
         let scribble = [ AnyShape (rectangle 10 20 5 6)
                        , AnyShape (circle 15 25 8)
                        ]

         -- Handle the shapes in the list polymorphically
         mapM_ (\s -> do draw' s
                         draw' (moveBy' s 100 100))
               scribble
```

Functional objects
with narrow operation for tail

```haskell
-- Narrow shapes to a uniform base type

narrowToShape :: Shape w -> Shape ()
narrowToShape s = s { setX       = narrowToShape . setX s
                    , setY       = narrowToShape . setY s
                    , moveTo     = \z -> narrowToShape . moveTo s z
                    , moveBy     = \z -> narrowToShape . moveBy s z
                    , shapeTail = () }


-- Test case for heterogeneous collections

main = do

      -- Construct a list of shapes
      let scribble = [ narrowToShape (rectangle 10 20 5 6)
                     , narrowToShape (circle 15 25 8)
                     ]

      -- Handle the shapes in the list polymorphically
      mapM_ (\s ->  do draw s
                       draw (moveBy s 100 100))
            scribble
```

Object encoding in Haskell: **Mutable objects**

```haskell
-- The type of mutable shapes

data Shape w =
     Shape { getX      :: IO Int
           , getY      :: IO Int
           , setX      :: Int -> IO ()
           , setY      :: Int -> IO ()
           , moveTo    :: Int -> Int -> IO ()
           , moveBy    :: Int -> Int -> IO ()
           , draw      :: IO ()
           , shapeTail :: w
           }
```

```haskell
-- The constructor for shapes

shape x y draw tail self
  = do
       xRef  <- newIORef x
       yRef  <- newIORef y
       tail'    <- tail
       return Shape
                 { getX       = readIORef xRef
                 , getY       = readIORef yRef
                 , setX       = writeIORef xRef
                 , setY       = writeIORef yRef
                 , moveTo     = \x y -> do { setX self x; setY self y }
                 , moveBy     = \dx dy ->
                                  do x <- getX self
                                     y <- getY self
                                     moveTo self (x+dx) (y+dy)
                 , draw       = draw self
                 , shapeTail = tail' self }
```

```haskell
-- Tail-polymorphic tails of circles

data CircleDelta w =
     CircleDelta { getRadius'  :: IO Int
                 , setRadius'  :: Int -> IO ()
                 , circleTail  :: w }


--- Circles as tail-instantiated shapes

type Circle w = Shape (CircleDelta w)


-- Convenient access to nested parts

getRadius = getRadius'  . shapeTail
setRadius = setRadius'  . shapeTail
```

```haskell
-- The constructor for circles

circle x y radius = shape x y draw tail
  where
    draw self
         =  do x <- getX self
               y <- getY self
               radius <- getRadius self
               putStrLn $ concat ["Drawing a Circle at:",
                                  show (x,y),
                                  ", radius ",
                                  show radius]

    tail = do rRef <- newIORef radius
              return (\self ->
                  CircleDelta { getRadius' = readIORef rRef
                              , setRadius' = writeIORef rRef
                              , circleTail = () })
```

```haskell
-- Construct a circle and draw it

testCircle =
  do
     c <- mfix $ circle 10 20 30
     draw c
```

```haskell
module Control.Monad.Fix where

{-

The fixed point of a monadic computation. mfix f executes the action f
only once, with the eventual output fed back as the input. Hence f
should not be strict, for then mfix f would diverge.

-}

class Monad m => MonadFix m
 where
  mfix :: (a -> m a) -> m a
```

```haskell
-- A variation on circle with logging facilities

circle' x y radius counter self
  = do super <- circle x y radius self
       return super
         { getX = do { tick; getX super }
         , getY = do { tick; getY super } }
 where
   tick = modifyIORef counter ((+) 1)


-- Construct a circle with logging and demonstrate it

testCircle' =
  do
     counterRef <- newIORef 0
     c <- mfix $ circle' 10 20 30 counterRef
     draw c
     counterVal <- readIORef counterRef
     putStrLn $ "#getter calls: " ++ show counterVal
```

Mutable objects
with existential quantification for tail

```haskell
-- Existential envelope for shapes

data AnyShape = forall x.
     AnyShape (Shape x)

draw'   (AnyShape s) = draw s
moveTo' (AnyShape s) = moveTo s
moveBy' (AnyShape s) = moveBy s


-- Test case for heterogeneous collections

main =
      do
         -- Construct a list of shapes
         s1 <- mfix $ rectangle 10 20 5 6
         s2 <- mfix $ circle 15 25 8
         let scribble = [ AnyShape s1
                        , AnyShape s2
                        ]

         -- Handle the shapes in the list polymorphically
         mapM_ (\s -> do draw' s
                         moveBy' s 100 100
                         draw' s)
               scribble
```

Mutable objects
with narrow operation for tail

```haskell
-- Narrow shapes to a uniform base type

narrowToShape :: Shape w -> Shape ()
narrowToShape s = s { shapeTail = () }


-- Test case for heterogeneous collections

main = do
         -- Construct a list of shapes
         s1 <- mfix $ rectangle 10 20 5 6
         s2 <- mfix $ circle 15 25 8
         let scribble = [ narrowToShape s1
                        , narrowToShape s2
                        ]

         -- Handle the shapes in the list polymorphically
         mapM_ (\s -> do draw s
                         moveBy s 100 100
                         draw s)
               scribble
```

Mutable objects

with the use of a heterogenous list

```haskell
-- Test case for heterogeneous collections

main = do
         -- Construct a list of shapes
         s1 <- mfix $ rectangle 10 20 5 6
         s2 <- mfix $ circle 15 25 8
         let scribble = ( s1, (s2, () ) )

         -- Handle the shapes in the list polymorphically
         myLoop scribble
```

```haskell
-- Model loop over collection as a type class

class MyLoop x
  where
    myLoop :: x -> IO ()

instance MyLoop ()
  where
    myLoop _ = return ()

instance MyLoop x => MyLoop (Shape w,x)
  where
    myLoop (s,x) = do draw s
                      moveBy s 100 100
                      draw s
                      myLoop x
```

Object encoding in Haskell

- Non-encapsulating encoding
- Encapsulating encoding
- **Composition-based & Interface-oriented**

```haskell
-- Data of shapes

data ShapeData =
     ShapeData { valX :: Int
               , valY :: Int }


-- The constructor for shapes

shape x y = ShapeData { valX = x
                      , valY = y }


-- The shape interface

class Shape s where
  getX, getY :: s -> Int
  setX, setY :: Int -> s -> s
  moveTo     :: Int -> Int -> s -> s
  moveTo x y =  setY y . setX x
  moveBy    :: Int -> Int -> s -> s
  moveBy deltax deltay s = moveTo x y s
   where x = getX s + deltax
         y = getY s + deltay
  draw       :: s -> IO ()
```

```haskell
-- The shape interface

class Shape s where
  getX, getY :: s -> Int
  setX, setY :: Int -> s -> s
  moveTo      :: Int -> Int -> s -> s
  moveTo x y =  setY y . setX x
  moveBy     :: Int -> Int -> s -> s
  moveBy deltax deltay s = moveTo x y s
   where x = getX s + deltax
         y = getY s + deltay
  draw        :: s -> IO ()

  -- Data-access convenience
  readShape  :: (ShapeData -> t)         -> s -> t
  writeShape :: (ShapeData -> ShapeData) -> s -> s
  getX        =  readShape valX
  getY        =  readShape valY
  setX i      =  writeShape (\s -> s  { valX = i })
  setY i      =  writeShape (\s -> s  { valY = i })
```

```haskell
-- The composed type of circles

data CircleData =
     CircleData { valShape  :: ShapeData
                , valRadius :: Int
                }


-- The constructor for circles

circle x y r
 = CircleData { valShape  = shape x y
              , valRadius = r
              }
```

```haskell
-- A circle is a shape

instance Shape CircleData
 where
  readShape f     = f . valShape
  writeShape f s = s { valShape = readShape f s }
  draw s
    =  putStrLn $ concat ["Drawing a Circle at:",
                          show (getX s,getY s),
                          ", radius ",
                          show (getRadius s)]
```

```haskell
-- The circle interface

class Shape s => Circle s
 where
  getRadius   :: s -> Int
  setRadius   :: Int -> s -> s

  -- Data-access convenience
  readCircle  :: (CircleData -> t)          -> s -> t
  writeCircle :: (CircleData -> CircleData) -> s -> s
  getRadius    =  readCircle valRadius
  setRadius i =  writeCircle (\s -> s  { valRadius = i })


-- A circle is circle is a ...

instance Circle CircleData
 where
  readCircle  = id
  writeCircle = id
```

Summary

- Folklore functional object encoding is of course possible.
- Existential quantification is in conflict with type inference.
- IO monad and IORefs enable imperative OO programming.
- Type classes could be used for interface-oriented programming.
- None of the ***shown*** options leads to convenient OOP model.