Simple Animations

- **Today's Topics**
  - Simple animations
  - Buffered graphics
  - Animations in Haskell
  - Complex animations
  - Lifting primitives to animations
  - Behaviors
  - Type classes, animations, and Behaviors
  - Time translation

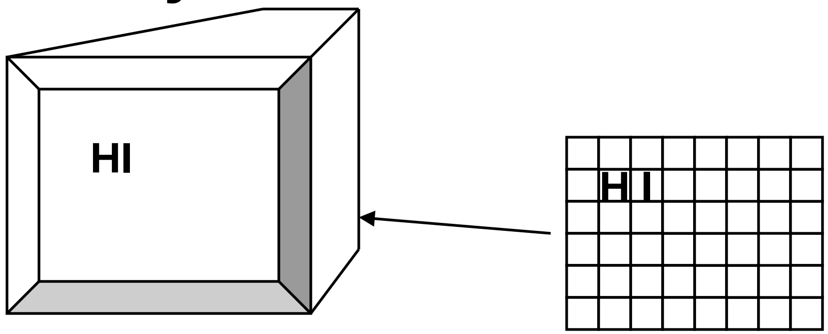
- **Reading Assignment**
  - Haskell School of Expression
    - Read chapter 13 - A Module of Simple Animations
-

Animations

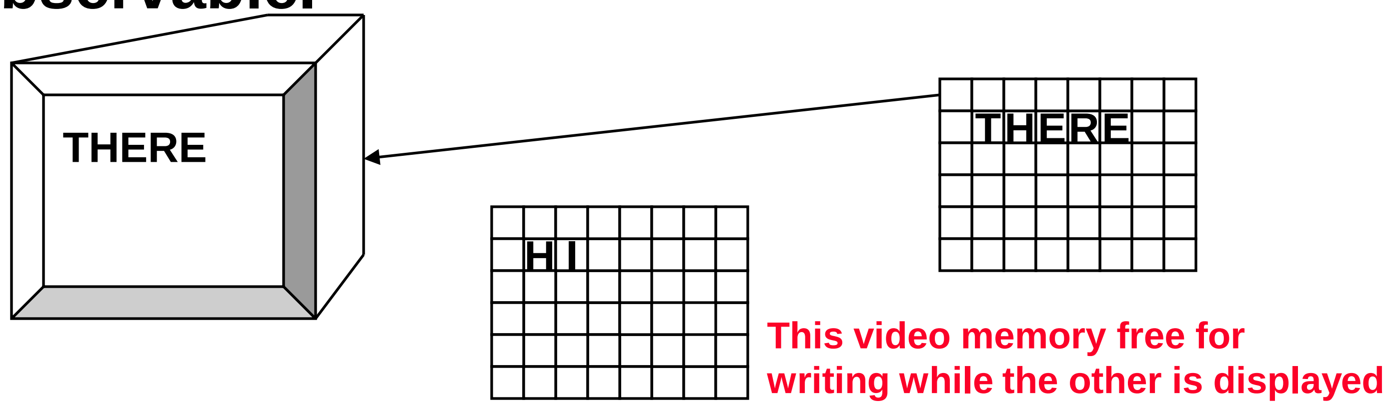
- An animation is a “moving” graphic.
  - Sometimes we say a time dependent graphic, since where it “moves” to is dependent upon time.
- To create the illusion of “movement” we need draw frames with a different picture each frame.
  - A frame rate of about 30 frames a second is optimal
  - less than 15-20 appears to flicker
  - greater than 30 gives no apparent improvement
- To draw a frame we need to erase the old frame before drawing the new frame.
- All our drawings have been accumulative (we never erase anything, just draw “over” what’s already there).
- There exist several strategies for frame drawing.

Buffered graphics

- **Display devices display the information stored in the video memory.**



- **Buffered graphics use two sets of memory, instantaneously switching from one memory to the other, so quickly that the flicker effect is unobservable.**



Haskell interface to buffered graphics

Usual tick rate = 30 times per second

- **timeGetTime**
  - `timegetTime :: IO Word32`
  - Returns the current time. This time has no real bearing on anything tangible. It is just a big number, and measures the time in milliseconds. The "difference" between successive calls accurately measures elapsed time.

- **setGraphic**
  - `setGraphic :: Window -> Graphic -> IO()`
  - Writes the graphic into the "free" video graphic buffer. At the next frame "tick" what's in the "free" video buffer will be drawn, and the current buffer will become the free buffer.

Interface to the richer window interface.

Old interface:

```
openWindow :: String -> Point -> IO Window
```

e.g. `openWindow "title" (width,height)`

Richer interface:

```
openWindowEx :: String -> Maybe Point
 Maybe Point -> (Graphic -> DrawFun) ->
  Maybe word32 -> IO Window
```

```
openWindowEx "title"
             (Just(x,y))    -- upper left corner
             (Just(width,height))
             drawFun
             (Just 30)      -- refresh rate
```

Animations in Haskell

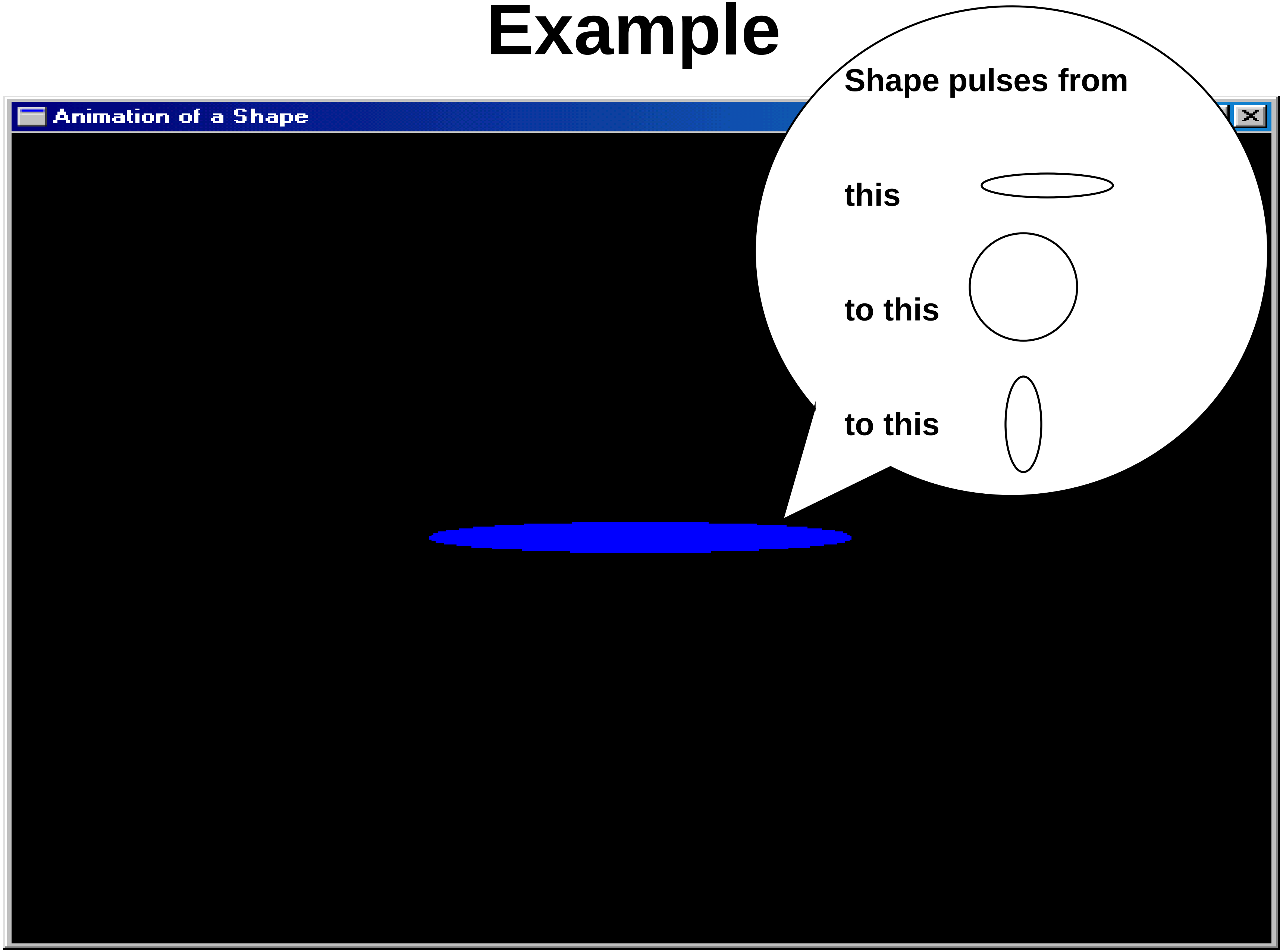
```
type Animation a = Time -> a
type Time = Float

rubberBall :: Animation Shape
rubberBall t = Ellipse (sin t) (cos t)

animate :: String -> Animation Graphic -> IO ()

main1 :: IO ()
main1 = animate "Animated Shape"
          (withColor Blue .
           shapeToGraphic .
           rubberBall)
```

Example



The animate function

```
animate :: String -> Animation Graphic -> IO ()

animate title anim
  = runGraphics (
    do w <- openWindowEx title (Just (0,0)) (Just Win,yWin))
                drawBufferedGraphic
       t0 <- timeGetTime
       let loop =
             do t <- timeGetTime
                let word32ToInt = fromInteger . toInteger
                let ft = intToFloat (word32ToInt(t-t0))/1000
                setGraphic w (anim ft)
                spaceCloseEx w loop
       loop
    )
```

Complex Animations

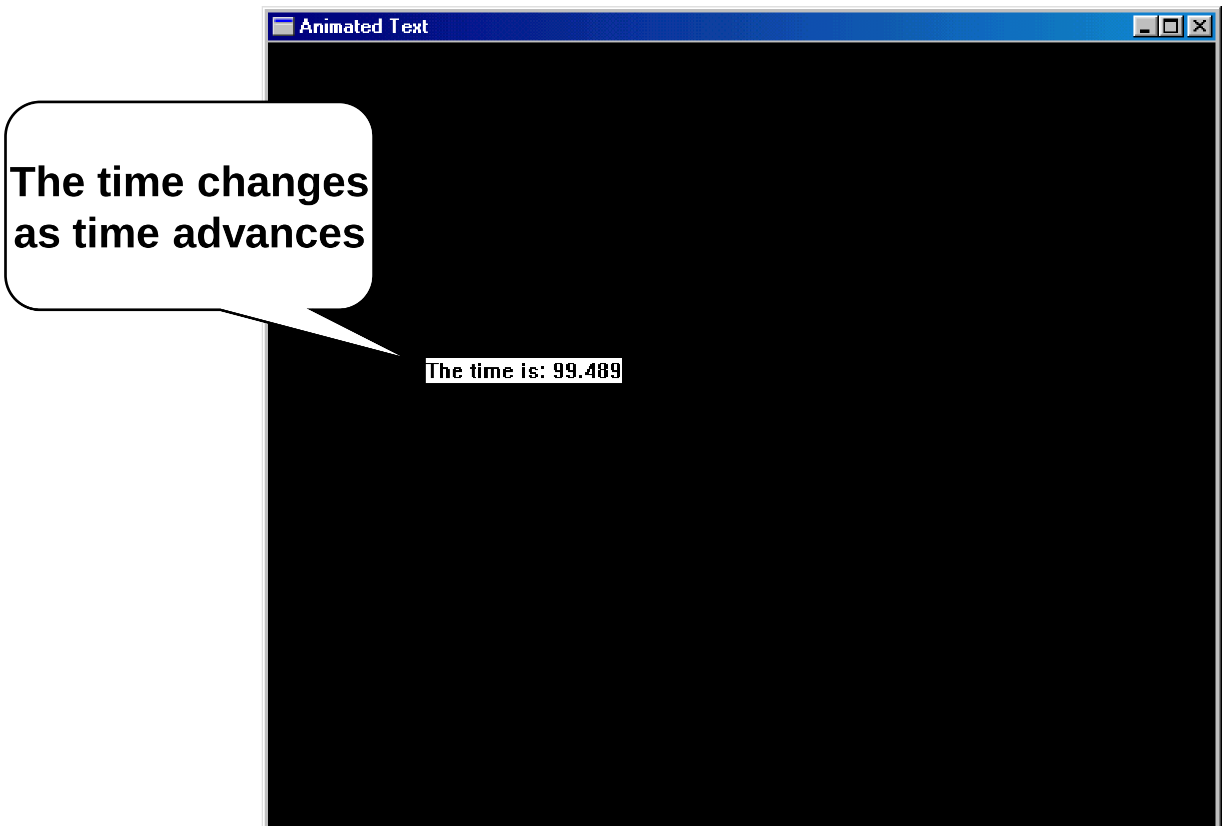
```
revolvingBallB :: Behavior Picture
revolvingBallB
  = let ball = shape (ell 0.2 0.2)
    in reg red (translate (sin time, cos time) ball)

planets :: Animation Picture
planets t
  = let p1 = Region Red (Shape (rubberBall t))
        p2 = Region Yellow (revolvingBall t)
    in p1 `Over` p2

tellTime :: Animation String
tellTime t = "The time is: " ++ show t
```

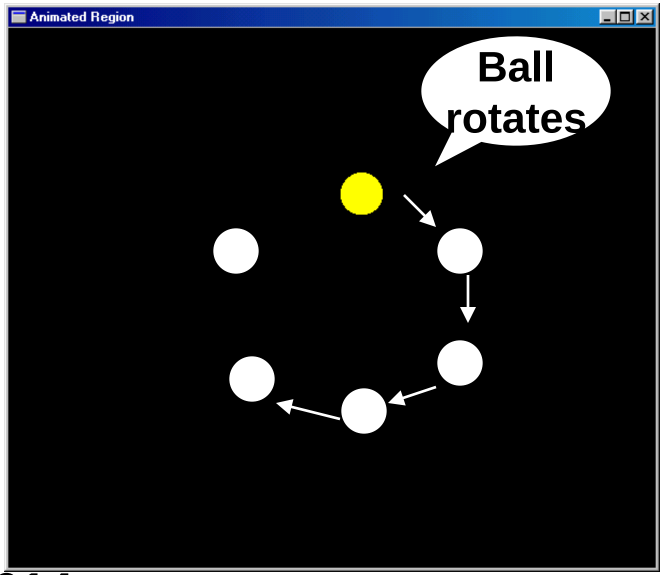
Telling Time

```
main2 = animate "Animated Text”
                tellTime
                (return . text (100,200))
```



Revolving Circle

```
regionToGraphic :: Region -> Graphic
regionToGraphic = drawRegion . regionToGRegion
main3 :: IO ()
main3 =
  animate "Animated Region"
    (withColor Yellow .
     regionToGraphic .
     revolvingBall)
```



Animating Pictures

```
picToGraphic :: Picture -> Graphic
picToGraphic (Region c r)
  = withColor c (regionToGraphic r)
picToGraphic (p1 `Over` p2)
  = picToGraphic p1 `overGraphic` picToGraphic p2
picToGraphic (Text v str) = (text (trans v) str)
picToGraphic EmptyPic = emptyGraphic

main4 :: IO ()
main4 = animate "Animated Picture"
              (picToGraphic . planets)
```

Case analysis over structure of region.

Use the primitives `overGraphic` & emptyGraphic

Lifting primitives to animations

- Its useful to define "time varying" primitives, like Picture

```
type Animation a = Time -> a
type Anim = Animation Picture
type Time = Float
```

- First an `Anim` which doesn't really vary

```
emptyA :: Anim
emptyA t = EmptyPic
```

Recall
`Anim =`
`Animation Picture =`
`Time -> Picture`
hence the time parameter t

- Combining time varying pictures

```
overA :: Anim -> Anim -> Anim
overA a1 a2 t = a1 t `Over` a2 t

overManyA :: [Anim] -> Anim
overManyA = foldr overA emptyA
```

Time Translation

```
timeTransA :: (Time -> Time) ->
                Animation a -> Animation a
```

or

```
timeTransA :: Animation Time ->
                Animation a -> Animation a
```

```
timeTransA f a t = a (f t)
```

or

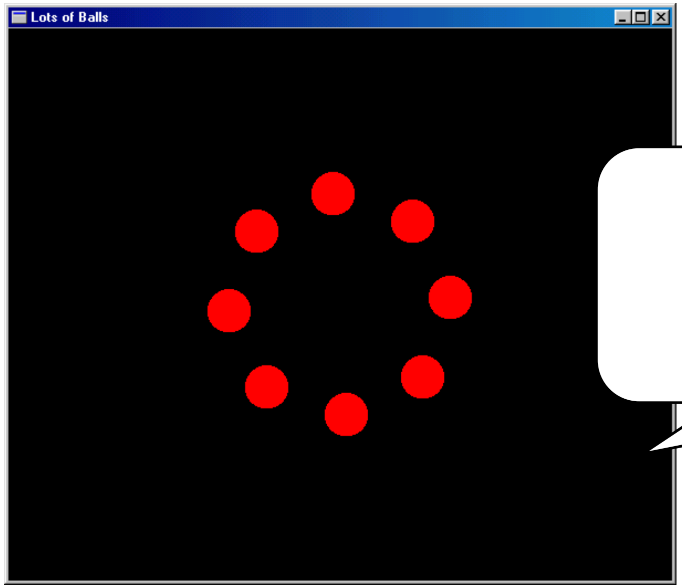
```
timeTransA f a = a . f
```

```
timeTransA (2*) anim    -- runs twice as fast
timeTransA (5+) anim    -- runs 5 seconds behind
```

Example

```
revolvingBallB :: Behavior Picture
revolvingBallB
  = let ball = shape (ell 0.2 0.2)
    in reg red (translate (sin time, cos time) ball)

main5 :: IO ()
main5 = animateB "Revolving Ball Behavior" revolvingBallB
```



Ball rotates around screen

Type Classes and Animations

- **"Polymorphism captures similar structure over different values, while type classes capture similar operations over different structure."**
- **Capture the similar operations on different things which vary over time with a Haskell Class.**
- **First define a new type:**

```
newtype Behavior a = Beh (Time -> a)
```

  - **`newtype` like `data` in Haskell**
  - **doesn't require the overhead that ordinary data definitions require since there is only 1 constructor function.**

Lifting ordinary functions to Behavior's

```
lift0 :: a -> Behavior a
lift0 x = Beh (\t -> x)

lift1 :: (a -> b) -> (Behavior a -> Behavior b)
lift1 f (Beh a) = Beh (\t -> f (a t))

lift2 :: (a -> b -> c) ->
         (Behavior a -> Behavior b -> Behavior c)
lift2 g (Beh a) (Beh b) = Beh (\t -> g (a t) (b t))

lift3 :: (a -> b -> c -> d) ->
 (Behavior a -> Behavior b -> Behavior c -> Behavior d)
lift3 g (Beh a) (Beh b) (Beh c)
   = Beh (\t -> g (a t) (b t) (c t))
```

Making Behavior Instances

```
instance Eq (Behavior a) where
  a1 == a2 = error "Can't compare animations."

instance Show (Behavior a)  where
   showsPrec n a1 =
     error "Can't coerce animation to String."
```

The instances for Eq and Show are bogus, but are necessary in order to define the Num class which requires Eq and Show

```
instance Num a => Num (Behavior a) where
  (+) = lift2 (+);  (*) = lift2 (*)
  negate = lift1 negate; abs = lift1 abs
  signum = lift1 signum
  fromInteger = lift0 . fromInteger
```

More Instances

```
instance Fractional a => Fractional (Behavior a)
  where
  (/) = lift2 (/)
  fromRational = lift0 . fromRational

instance Floating a => Floating (Behavior a) where
   pi    = lift0 pi;    sqrt = lift1 sqrt
   exp   = lift1 exp;   log = lift1 log
   sin   = lift1 sin;   cos = lift1 cos
   tan   = lift1 tan
   asin  = lift1 asin;  acos = lift1 acos
   atan  = lift1 atan
   sinh  = lift1 sinh;  cosh = lift1 cosh
   tanh  = lift1 tanh
   asinh = lift1 asinh; acosh = lift1 acosh
   atanh = lift1 atanh
```

Time

```
time :: Behavior Time
time = Beh (\t -> t)
```

A New Class

```
class Ani a where
  empty :: a
  over  :: a -> a -> a
```

Instances for Our types

```
instance Ani [a] where
  empty = []
  over  = (++)

data Fun a = Fun (a->a)
instance Ani (Fun a) where
  empty = Fun id
  Fun a `over` Fun b = Fun (a . b)

instance Ani Picture where
  empty = EmptyPic
  over  = Over

instance Ani a => Ani (Behavior a) where
  empty = lift0 empty
  over  = lift2 over
```

What type is "empty" here?

Things that can turn

```
class Turnable a where
  turn :: Float -> a -> a

instance Turnable Picture where
  turn theta (Region c r) =
       Region c (turn theta r)  -- turn on Regions
  turn theta (p1 `Over` p2) = turn theta p1 `Over`
  turn theta p2
  turn theta EmptyPic = EmptyPic

instance Turnable a => Turnable (Behavior a) where
  turn theta (Beh b) = Beh(turn theta . b)
```

Turning Shapes

```
type Coordinate = (Float,Float)

rotate :: Float -> Coordinate -> Coordinate
rotate theta (x,y) =
    (x*c + y*s, y*c - x*s)
    where (s,c) = (sin theta,cos theta)

instance Turnable Shape where
  turn theta (Polygon ps) =
       Polygon (map (rotate theta) ps)
  -- lots of missing cases here for
  -- turn theta (Rectangle s1 s2)  =
  -- turn theta (Ellipse r1 r2)    =
  -- turn theta (RtTriangle s1 s2) =
```

Turning Regions

```
instance Turnable Region where
  turn theta (Shape sh) = Shape (turn theta sh)
   -- lots of missing cases here for
   -- turn theta (Translate (u,v) r)   =
   -- turn theta (Scale (u,v) r)       =
   -- turn theta (Complement r)        =
   -- turn theta (r1 `Union` r2)       =
   -- turn theta (r1 `Intersect` r2)   =
   -- turn theta Empty = Empty
```

A final example. See the text pages 209-212 main7 in today's Haskell code.