Motivation

- In the abstract, an *animation* is a continuous, time-varying image.
- In practice, it is a sequence of static images displayed in succession so rapidly that it looks continuous.
- Our goal is to present to the programmer an abstract view of animations that hides the practical details.
- In addition, we will generalize animations to be continuous, time-varying quantities of any value, not just images.

Representing Animations

- As usual, we will use our most powerful tool, *functions*, to represent animations:
  
  ```hs
  type Animation a = Time -> a
  type Time = Float
  ```

- Examples:
  ```hs
  rubberBall :: Animation Shape
  rubberBall t = Ellipse (sin t) (cos t)
  revolvingBall :: Animation Region
  revolvingBall t = let ball = Shape (Ellipse 0.2 0.2)
                      in Translate (sin t, cos t) 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
  ```

An Animator

- Given a function...
  ```hs
  animate :: String -> Animation Graphic -> IO ()
  ```
  ...we could then execute (display) the previous animations like this:

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

  main2 :: IO ()
  main2 = animate "Animated Text"
           (text (100,200) . tellTime)
  ```

 Definition of “animate”

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

  ```hs
  animate title anim = runGraphics $ do
  w <- openWindowEx title (Just (0,0)) (Just (xWin,yWin))
  drawBufferedGraphic (Just 30)
  t0 <- timeGetTime
  let loop = do
  t <- timeGetTime
  let ft = intToFloat (word32ToInt (t-t0)) / 1000
  setGraphic w (anim ft)
  getWindowTick w
  loop
  loop
  ```

  See text for details...

Common Operations

- We can define many operations on animations based on the underlying type. For example, for Pictures:
  ```hs
  emptyP :: Animation Picture
  emptyP t = EmptyPic
  ```

- We can do a similar thing for Shapes, etc.
- Also, for numeric animations, we could define functions like addA, multA, and so on.
- But there is a better way...
Behaviors

- Basic definition (replacing Animation):
  \[
  \text{newtype Behavior } a = \text{Beh} \ (\text{Time} \rightarrow a)
  \]
  
- Recall that \text{newtype} creates a single-argument datatype with (time and space) efficiency the same as a simple \text{type} declaration. (So then what is the difference??)

Behaviors

- We need to use \text{newtype} here because type synonyms are not allowed in type class instance declarations -- only types declared with \text{data} of \text{newtype}.

Constant Behaviors

- Given a scalar value \( x \), we can lift it to a constant behavior that, at all times \( t \), yields \( x \):
  \[
  \text{lift0} :: a \rightarrow \text{Behavior } a
  \]
  \[
  \text{lift0 } x = \text{Beh} \ (\lambda t \rightarrow x)
  \]

Dependent Behaviors

- Given a function \( f \), we can lift it to a function on behaviors that, at a given time \( t \), samples its argument and passes the result through \( f \):
  \[
  \text{lift1} :: (a \rightarrow b) \rightarrow (\text{Behavior } a \rightarrow \text{Behavior } b)
  \]
  \[
  \text{lift1 } f \ (\text{Beh } a) = \text{Beh} \ (\lambda t \rightarrow f \ (a \ t))
  \]

Numeric Behaviors

- \text{instance Num } a =
  \[
  > \text{Num} \ (\text{Behavior } a)
  \]
  \[
  \text{where}
  \]
  \[
  (+) = \text{lift2 } (+)
  \]
  \[
  (*) = \text{lift2 } (*)
  \]
  \[
  \text{negate} = \text{lift1 } \text{negate}
  \]
  \[
  \text{abs} = \text{lift1 } \text{abs}
  \]
  \[
  \text{signum} = \text{lift1 } \text{signum}
  \]
  \[
  \text{fromInteger} = \text{lift0} \ . \ \text{fromInteger}
  \]

Type Class Magic

- Furthermore, define \text{time} as a behavior:
  \[
  \text{time} :: \text{Behavior } \text{Time}
  \]
  \[
  \text{time} = \text{Beh} \ (\lambda t \rightarrow t)
  \]

- Now consider the expression “\text{time + 42}”:
  \[
  \text{time} + 42
  \]
  \[
  \rightarrow \text{unfold overloaded def of } \text{time}, (+), \text{and 42}
  \]
  \[
  (\text{lift2 } (+)) \ (\text{Beh} \ (\lambda t \rightarrow t)) \ (\text{Beh} \ (\lambda t \rightarrow 42))
  \]
  \[
  \rightarrow \text{unfold lift2}
  \]
  \[
  (\lambda \ (\text{Beh } a) \ (\text{Beh } b) \rightarrow \text{Beh} \ (\lambda t \rightarrow a \ t + b \ t)) \ (\text{Beh} \ (\lambda t \rightarrow t)) \ (\text{Beh} \ (\lambda t \rightarrow 42))
  \]
  \[
  \rightarrow \text{perform some anonymous function applications}
  \]
  \[
  \text{Beh} \ (\lambda t \rightarrow (t \rightarrow t + 42) \ t)
  \]
  \[
  \rightarrow \text{and NO WANK}
  \]
  \[
  \text{Beh} \ (\lambda t \rightarrow t + 42)
  \]

this is cool
New Type Classes

- Besides lifting existing type classes such as \texttt{Num} to behaviors, we can define new classes for manipulating behaviors. For example:

  ```hs
  class Combine a where
      empty :: a
      over :: a -> a -> a
  instance Combine Picture where
      empty = EmptyPic
      over = Over
  instance Combine a => Combine (Behavior a) where
      empty = lift0 empty
      over = lift2 over
  overMany :: Combine a => [a] -> a
  overMany = foldr over empty
  ```

Hiding More Detail

- We have not yet hidden all the “practical” details of animation — in particular time itself.
- But through more aggressive lifting...

  ```hs
  reg = lift2 Region
  shape = lift1 Shape
  all = lift2 Ellipse
  red = lift0 Red
  yellow = lift0 Yellow
  translate (Beh x, Beh y) = (x, y) -- one version new
  translate (Beh x, Beh y) (x', y') = (Beh x', Beh y') -- another version
  ...
  ```

- we can redefine our red revolving ball without referring to time at all:

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

More Liftings

- Comparison operators:

  ```hs
  (>*) :: Ord a => Behavior a -> Behavior a -> Behavior Bool
  (>*) = lift2 (>)
  ```

- Conditional behaviors:

  ```hs
  cond :: Behavior Bool -> Behavior a -> Behavior a
  cond = lift3 (\p c a -> if p then c else a)
  ```

- For example, a flashing color:

  ```hs
  flash :: Behavior Color
  flash = cond (sin time >* 0) red yellow
  ```

Time Travel

- A function for translating a behavior through time:

  ```hs
  timeTrans :: Behavior Time -> Behavior a -> Behavior a
  timeTrans (Beh f) (Beh a) = Beh (a . f)
  ```

  ```hs
  For example:
  ```

  ```hs
  timeTrans (2*time) anim
  timetrans (5+time) anim `over` anim
  timeTrans (negate time) anim
  ```

- Any kind of behavior can be time transformed:

  ```hs
  flashingBall :: Behavior Picture
  flashingBall =
      let
          ball = shape (ell 0.2 0.2)
      in
          reg (timeTrans (8*time) flash)
          (translate (sin time, cos time) ball)
  ```

Final Example

- A kaleidoscope program. (The details of its construction can be skimmed, but you may enjoy running it...)

  ```hs
  revolvingBalls :: Behavior Picture
  revolvingBalls =
      overMany (timeTrans (time + t*pi/4) flashingBall)
      [ 0 .. map lift0 [0..7] ]
  ```

See SOE for a more substantial example: a kaleidoscope program.