Advanced Programming Handout 10

A Module of Simple Animations

(SOE Chapter 13)

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:

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

- Examples:

```
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...

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

...we could then execute (display) the previous animations like this:

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

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

Definition of “animate”

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

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:

```
emptyA :: Animation Picture
emptyA t = EmptyPic

overA :: Animation Picture
         -> Animation Picture
         -> Animation Picture
overA a1 a2 t = a1 t `Over` a2 t

overManyA :: [Animation Picture] -> Animation Picture
overManyA = foldr overA emptyA
```

- 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...

Common Operations

- We can define many operations on animations based on the underlying type. For example, for Pictures:

```
emptyA :: Animation Picture
emptyA t = EmptyPic

overA :: Animation Picture
         -> Animation Picture
         -> Animation Picture
overA a1 a2 t = a1 t `Over` a2 t

overManyA :: [Animation Picture] -> Animation Picture
overManyA = foldr overA emptyA
```

- 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...

Type Classes!

(naturally)

Behaviors

- Basic definition (replacing `Animation`):

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

- Recall that `newtype` creates a single-argument datatype with (time and space) efficiency the same as a simple `type` declaration.

(So then what is the difference??)

Behaviors

- We need to use `newtype` here because type synonyms are not allowed in type class instance declarations -- only types declared with `data` or `newtype`.

Constant Behaviors

- Given a scalar value **`x`**, we can lift it to a constant behavior that, at all times **`t`**, yields **`x`**:

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

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

Numeric Behaviors

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

```
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
etc.
```

...and similarly for the other basic classes (Fractional, etc.)

...where:

```
 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))
```

Type Class Magic

- Furthermore, define *time* as a behavior:

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

- Now consider the expression "`time + 42`":

```
time + 42
```

➔ unfold overloaded defs of time, (+), and 42

```
(lift2 (+)) (Beh (\t -> t)) (Beh (\t -> 42))
```

➔ `unfold lift2`

```
(\ (Beh a) (Beh b) -> Beh (\t -> a t + b t) )
      (Beh (\t -> t))
      (Beh (\t -> 42))
```

➔ perform some anonymous function applications

```
Beh (\t -> (\t -> t) t + (\t -> 42) t )
```

➔ and two more

```
Beh (\t -> t + 42)
```

this is cool

New Type Classes

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

```
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...

```
reg     = lift2 Region
shape   = lift1 Shape
ell     = lift2 Ellipse
red     = lift0 Red
yellow  = lift0 Yellow
translate (Beh a1, Beh a2) (Beh r)           -- note subtlety here
        = Beh (\t -> Translate (a1 t, a2 t) (r t))
```

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

```
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:

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

- Conditional behaviors:

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

- For example, a flashing color:

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

Time Travel

- A function for translating a behavior through time:

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

- For example:

```
timeTrans (2*time) anim                      -- double speed
(timeTrans (5+time) anim) `over` anim        -- one anim 5 sec
                                                behind another
timeTrans (negate time) anim                 -- go backwards
```

- Any kind of behavior can be time transformed:

```
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

```
revolvingBalls :: Behavior Picture

revolvingBalls =
   overMany [ timeTrans (time + t*pi/4) flashingBall
            | t <- map lift0 [0..7] ]
```

See SOE for a more substantial example: a kaleidoscope program. *(The details of its construction can be skimmed, but you may enjoy running it...)*