A Domain-Specific Language for Animation

COS 441 Slides 8

Slide content credits:
Paul Hudak's School of Expression
Ranjit Jhala (UCSD)
The last few weeks
  - the principles of functional programming
    • defining new functions: functional abstraction for code reuse
    • defining new types: type abstraction
    • higher-order programming: using functions as data
    • the same algorithm over different data: parametric polymorphism
    • related operations over different types: ad hoc polymorphism via type classes

This time:
  - Bringing it all together: developing a domain-specific language for functional animation
SHAPES, REGIONS & PICTURES
data Shape =
  Rectangle Side Side
  | Ellipse Radius Radius
  | RtTriangle Side Side
  | Polygon [Vertex]
deriving (Show)

type Side = Float
type Radius = Float
type Vertex = (Float, Float)
data Shape =
  Rectangle Side Side
  | Ellipse Radius Radius
  | RtTriangle Side Side
  | Polygon [Vertex]
deriving (Show)

    type Side = Float
    type Radius = Float
    type Vertex = (Float, Float)

s1 = Rectangle 3 2
s2 = Ellipse 1 1.5
s3 = RtTriangle 3 2
s4 = Polygon [(-2.5, 2.5),
               ,(-3, 0),
               ,(-1.7,-1.0),
               ,(-1.1,0.2),
               ,(-1.5,2.0) ]
Regions

- Regions are compositions of basic shapes:

  data Region =
  \hspace{1cm} Shape Shape -- primitive shape
  | Translate Vector Region -- translated region
  | Scale Vector Region -- scaled region
  | Complement Region -- inverse of region
  | Region `\text{\textbackslash Union}` Region -- union of regions
  | Region `\text{\textbackslash Intersect}` Region -- intersection of regions
  | Region `\text{\textbackslash Xor}` Region -- XOR of regions
  | Empty -- empty region

  deriving Show

  type vector = (Int, Int)
Regions

- Regions are compositions of basic shapes:

```haskell
data Region =
  Shape Shape -- primitive shape
  | Translate Vector Region -- translated region
  | Scale Vector Region -- scaled region
  | Complement Region -- inverse of region
  | Region `Union` Region -- union of regions
  | Region `Intersect` Region -- intersection of regions
  | Region `Xor` Region -- XOR of regions
  | Empty -- empty region

deriving Show

type vector = (Int, Int)

r1 = Shape s1
r2 = Shape s2
r3 = Shape s3
r4 = Shape s4
reg0 = (Complement r2) `Union` r4
reg1 = r3 `Union` (r1 `Intersect` r0)
```
Regions

• Notice that regions are recursive data structures; consequently, they can be arbitrarily complex:

\[
\text{step} = \text{Shape} \left( \text{Rectangle} \ 50 \ 50 \right)
\]

\[
\text{stairs} \ k = \\
\quad \text{if} \ k \leq 0 \ \text{then} \ \text{Empty} \\
\quad \text{else} \ \text{Translate} \ (k \times 20, k \times 20) \ (\text{step} \ \text{`Union`} \ \text{stairs} \ (k - 1))
\]

stairs 4 =
Pictures

• Pictures add color to regions

```haskell
data Picture =
    Region Color Region
  | Picture `Over` Picture
  | EmptyPic
deriving Show

type Color = Red | Yellow | ...
```

• Some pictures:

```haskell
pic1 = Region Red reg1
r5 = Shape $ Rectangle 1 1
r6 = Shape $ Ellipse 0.5 0.5
reg2 = (Scale (2,2) r6) `Union` (Translate (2,1) r6) `Union` (Translate (-2,0) r5)
pic2 = Region Yellow reg2
pic3 = pic2 `Over` pic1
```
the SOE libraries have implemented a draw function for us:

```
type Title = String
draw :: Title -> Picture -> IO ()
```

try it:

```
main1 = draw "Picture 1" pic1
main2 = draw "Picture 2" pic2
main3 = draw "Picture 3" pic3
```

go to demo
FROM STATIC PICTURES
TO DYNAMIC ANIMATIONS
Animation

• We create animations by exploiting persistence of vision and rendering a series of images:
  1. Initialize image
  2. Render image
  3. Pause
  4. Change image
  5. Go to 1.

• At a low level, this is what will happen, but we'd like to build a library of combiners (ie: functions) that can be reused and that allow us to build complex animations from simpler parts
We are going to represent an animation using a function:

\[
\text{type Animation}\ a = \text{Time} \to a
\]

\[
\text{type Time} = \text{Float}
\]

At every instant in time, the animation function generates an object with type \( a \).

Since the animation type is polymorphic, we'll be able to animate many different kinds of things:

\[
\text{type PictureAnimation} = \text{Time} \to \text{Picture}
\]
\[
\text{type ShapeAnimation} = \text{Time} \to \text{Shape}
\]
\[
\text{type StringAnimation} = \text{Time} \to \text{String}
\]
A first animation

- Once you've thought of the right type, defining basic animations is easy:

rubberBall :: Animation Shape
rubberBall = \( t \rightarrow \text{Ellipse} (\sin t) (\cos t) \)
More Animations

revolvingBall :: Animation Region

revolvingBall = \t -> Translate (sin t, cos t) ball
where ball = Shape (Ellipse 0.2 0.2)
• Composition at work!
• By making animations functions, we can compose them using ordinary function application or function composition:

rubberBall :: Animation Shape
rubberBall = \( t \rightarrow \text{Ellipse}(\sin t)(\cos t) \)

revolvingBall :: Animation Region
revolvingBall = \( t \rightarrow \text{Translate}(\sin t, \cos t)\) ball
where ball = Shape(Ellipse 0.2 0.2)

planets :: Animation Picture
planets t = p1 `Over` p2
where p1 = Region Red $ Shape(rubberBall t)
p2 = Region Yellow $ revolvingBall t
More Animations

• We can animate anything:


ticker :: Animation String

ticker t = "The time is :" ++ show t

• An animation is any time-varying value
A **Graphic** is a data structure representing a static picture that can be rendered efficiently.

To render any animation, we need two things:
- A function to convert an **Animation a** to an **Animation Graphic**
- A function to render any **Animation Graphic**

The second is supplied by the SOE library:

```haskell
animate :: Title -> Animation Graphic -> IO ()
```

The first can be developed provided we have some basic **Graphic generators**:

```haskell
shapeToGraphic :: Shape -> Graphic
regionToGraphic :: Region -> Graphic
pictureToGraphic :: Picture -> Graphic
text :: Point -> String -> Graphic
withColor :: Color -> Graphic -> Graphic
```
A simple example:

\[
\text{blueBall} :: \text{Animation Graphic}
\]
\[
\text{blueBall} = \text{withColor Blue . shapeToGraphic . rubberBall}
\]

Check: does it have the right type?

\[
\text{rubberBall} :: \text{Time -> Shape}
\]
\[
\text{shapeToGraphic} :: \text{Shape -> Graphic}
\]
\[
\text{withColor Blue} :: \text{Graphic -> Graphic}
\]
\[
\text{withColor Blue . shapeToGraphic . rubberBall} :: \text{Time -> Graphic}
\]
\[
\text{= Animation Graphic}
\]

Let's try to run it
• Let's look at some more:

\[
\text{main4} = \text{animate } \"\text{Shape}\" \$ \text{withColor Blue} . \text{shapeToGraphic} . \text{rubberBall}
\]

\[
\text{main5} = \text{animate } \"\text{Text}\" \$ \text{text (100,200)} . \text{ticker}
\]

\[
\text{main6} = \text{animate } \"\text{Region}\" \$ \text{withColor Yellow} . \text{regionToGraphic} . \text{revolvingBall}
\]

\[
\text{main7} = \text{animate } \"\text{Picture}\" \$ \text{picToGraphic} . \text{planets}
\]
Implementing Animate

- Some details of the animator (see script for more):

```haskell
animate title anim = runGraphics $ do
  w <- openWindowEx title (Just (0,0)) (Just (xWin, yWin)) drawBufferedGraphic
  t0 <- timeGetTime
  animateLoop w t0 anim

animateLoop w t0 anim = do
  t <- timeGetTime
  let ft = intToFloat (fromInteger (toInteger (t - t0))) / 1000
  setGraphic w (anim ft)
  spaceCloseEx w $ animateLoop w t0 anim
```

- Set up window
- Begin animation loop with initial time
- Compute next time
- Draw the picture at the computed time
- Check for termination signal
- Continue
GOING FURTHER:
A DSL FOR ANIMATIONS
An Embedded DSL for Animations

• So far, we've built animations bottom-up with $\text{Time} \to \text{a}$ functions

• But:
  – we can't (easily) transform or modify existing animations
  – we can't (easily) compose existing, fully-formed animations
  – we don't treat animations as abstract objects

• The next step:
  – Treat animations as abstract objects and define canonical transformers for them
  – Work entirely at the level of animations, hiding the implementation details
  – Our implementation might be called "a cool library" but ... we hide the underlying details so thoroughly we'll call the library an embedded, domain-specific language.
  – Haskell, with it's lightweight syntax and facilities for reuse and abstraction, is a terrific platform for developing new DSLs
DSL Design Strategy

• Choose primary abstract objects
  – define special types to represent them
  – in our case: a special abstract Behavior type

• Define operations over the abstract objects
  – make the above abstract objects instances of well-chosen type classes where appropriate so we can use compact, intuitive notation for manipulating our objects
  – in our case: make behaviors instances of type classes for graphical and numeric manipulation
type Behavior a

type Coordinates = (Behavior Float, Behavior Float)

run :: Behavior Picture -> IO ()

red :: Behavior Color

ell :: Behavior Radius -> Behavior Radius -> Behavior Shape

shape :: Behavior Shape -> Behavior Region

reg :: Behavior Color -> Behavior Region -> Behavior Picture

over :: Behavior Picture -> Behavior Picture -> Behavior Picture

sin :: Behavior Float -> Behavior Float

tx :: Coordinates -> Behavior Picture -> Behavior Picture

timeTx :: Behavior Time -> Behavior a -> Behavior a

rewind :: Behavior a -> Behavior a

lift0 :: a -> Behavior a

lift1 :: (a -> b) -> Behavior a -> Behavior b

lift2 :: (a -> b -> c) -> Behavior a -> Behavior b -> Behavior c
Examples

• A stationary ball:

  \[ \text{demo1} = \text{run \$ reg yellow \$ ballB} \]

• Bouncing the ball:

  \[ \text{demo2} = \text{run \$ reg yellow \$ tx (0, \sin \text{time}) ballB} \]

• Bouncing a triangle:

  \[ \text{demo2} = \text{run \$ reg yellow \$ tx (0, \sin \text{time}) pentaB} \]

• Bouncing anything yellow:

  \[ \text{bounce b} = \text{reg yellow \$ tx (0, \sin \text{time}) b} \]
Examples

• Colors can vary with time. Why stick with constant yellow?

  flash :: Behavior Color

  demo4 = run $ reg flash $ tx (0, sin time) ballB

• Any animation can be composed with any other

  demo5 = run $ a1 `over` a2
  where a1 = reg red $ tx (0, sin time) ballB
  a2 = reg yellow $ tx (sin time, 0) pentaB
Examples

- We can define new kinds of motions and apply them to many different kinds of objects

```haskell
turn :: (Deformable a) => Float -> a -> a
lift2 :: (a -> b -> c) -> Behavior a -> Behavior b -> Behavior c
lift2 turn :: Behavior Float -> Behavior a -> Behavior a

demo6 = run $ a1 `over` a2
  where a1 = reg red $ tx (0, sin time) ballB
        a2 = reg yellow $ lift2 turn angle pentaB
        angle = pi * sin time
```

angle is a behavior.
notice the overloading:
type classes!
Examples

- We can manipulate time itself! Thereby delaying, slowing down or speeding up animations.

```plaintext
demo7 = run $ a1 `over` a2
   where a1 = reg red $ tx (sin time, cos time) ballB
   a2 = timeTx (2 + time) a1
```

notice the overloading: type classes!

```plaintext
demo8 = run $ a1 `over` a2
   where a1 = reg red $ tx (sin time, cos time) ballB
   a2 = timeTx (2 * time) a1
```

a delayed animation composed with itself

a fast-forwarded animation
Examples

• We can even put time in reverse and run an animation backwards. (Makes me wonder if we could do some DVR programming in Haskell ...)

demo0 = run $ a1 `over` a2
  where a1 = reg red $ tx (sin time, cos time) ballB
        a2 = timeTx (-1 * time) a1
run backwards
BUILDING THE DSL
Whereas an animation was just a synonym for a function type, a behavior is abstract:

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

There are a couple of reasons:

- we would like to control the invariants governing Behaviors
- we would like to hide implementation details from clients
- we will be using some type classes, and type classes don't work properly with type synonyms
  - why? Intuitively because a synonym is completely interchangeable with its definition. Hence, we can't define a different behavior for the synonym than its definition. (If we could, they wouldn't be interchangeable.)

Note: A newtype is a data type with just 1 constructor and no performance overhead for using it
newtype Behavior a = Beh Time -> a

animateB :: String -> Behavior Picture -> IO ()
animateB s (Beh f) = animate s (picToGraphic . f)

run = animateB "Animation Window"
Bootstrapping

- Recall the map function: It took an ordinary function and made it into a function over lists:

  \[ \text{map} :: (a \rightarrow b) \rightarrow ([a] \rightarrow [b]) \]

- One might say that map "lifts" an ordinary function up in to the domain of list-processing functions

- Likewise, we will want to "lift" ordinary functions up in to the domain of behavior-processing functions:

  \[ \text{lift1} :: (a \rightarrow b) \rightarrow \text{Behavior} \ a \rightarrow \text{Behavior} \ b \]
  \[ \text{lift1} \ f \ (\text{Beh} \ g) = \text{Beh} \ (\lambda t \rightarrow f \ (g \ t)) \]

- Lift is a way to include all of Haskell's powerful function-definition facilities within our newly developed DSL
• Lift1 works with single-argument functions. We may need to do heavier lifting:

lift2 :: (a -> b -> c) -> Behavior a -> Behavior b -> Behavior c
lift2 f (Beh a) (Beh b) = Beh $ \_t -> f (a t) (b t)

lift3 :: (a -> b -> c -> d) -> Behavior a -> Behavior b -> Behavior c -> Behavior d
lift3 f (Beh a) (Beh b) (Beh c) = Beh $ \_t -> f (a t) (b t) (c t)

• You can think of a constant, like the color Red, as a 0-argument function. We'll want to lift constants too:

lift0 :: a -> Behavior a
lift0 x = Beh $ \_t -> x

a constant function; it returns x all the time
Bootstrapping

• Since lists are so common in Haskell, we'll lift list-processing functions too

• Explore the details in your spare time:

  liftXs :: ([t] -> a) -> [Behavior t] -> Behavior a
  liftXs f bs = Beh (\t -> f (map (\(Beh b) -> b t) bs))

• But notice, even without looking at the code, how much information you get out of the type of the function:

  liftXs :: ([t] -> a) -> ([Behavior t] -> Behavior a)

• There's really only 1 reasonable thing that liftXs could do, given its type
Numeric Behaviors

- Our examples involve managing coordinates, scaling factors and timewarp; we need support for numeric behaviors
- Let's define standard numeric operations over behaviors by making it an instance of the Num Class

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

• Unsure what (+) on Behaviors does? Run through an example using computation by calculation

instance Num a => Num (Behavior a) where
  (+) = lift2 (+) ... 

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

lift0 :: a -> Behavior a
lift0 x = Beh $ \t -> x

one = Beh (\t -> 1)
time = Beh (\t -> t)

(+) time one
= lift2 (+) time one
= lift2 (+) (Beh (\t -> t)) (Beh (\t -> 1))
= Beh (\t -> (+) ((\t -> t) t) ((\t -> 1) t))
= Beh (\t -> (+) t 1)
= Beh (\t -> t + 1)

It just adds the numbers from the same time instant!
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
Once again, check our work by calculating

\[ \text{instance Floating a} \Rightarrow \text{Floating (Behavior a)} \text{ where} \\
\quad \sin = \text{lift1} \sin \\
\quad \ldots \\
\]

\[ \text{lift1} :: (a \rightarrow b) \rightarrow \text{Behavior a} \rightarrow \text{Behavior b} \\
\quad \text{lift1} f (\text{Beh } g) = \text{Beh } (\lambda t \rightarrow f (g t)) \\
\]

\[ \text{time} :: \text{Behavior Time} \\
\quad \text{time} = \text{Beh } (\lambda t \rightarrow t) \\
\]

\[ \sin \text{ time} = \text{lift1} \sin \text{ time} \\
\quad = \text{lift1} \sin (\text{Beh } (\lambda t \rightarrow t)) \\
\quad = \lambda t \rightarrow \sin ((\lambda t \rightarrow t) \ t) \\
\quad = \lambda t \rightarrow \sin t \]
Add in Operations for Colors, Pictures, Regions

reg = lift2 Region
shape = lift1 Shape
poly = liftXs Polygon
ell = lift2 Ellipse
red = lift0 Red
yellow = lift0 Yellow
green = lift0 Green
blue = lift0 Blue

tx (Beh a1, Beh a2) (Beh r) = Beh (\t -> Translate (a1 t, a2 t) (r t))

- Ok, at this point, you've got to admit that whoever came up with the concept of "lifting" and the idea of defining the liftN functions was pretty smart -- they are getting a lot of play!
Creating Behavioral Shapes

• Our basic ball:

\[
\begin{align*}
\text{ballB} &: \text{Behavior Region} \\
\text{ballB} &= \text{shape } \ell 0.2 0.2
\end{align*}
\]

• Our basic pentagon:

\[
\begin{align*}
\text{pentaB} &: \text{Behavior Region} \\
\text{pentaB} &= \text{shape } \text{poly} (\text{map lift0 vs}) \\
\text{where vs} &= [ (0.0, 0.8) \\
&\quad , (0.3,-0.5) \\
&\quad , (-0.3,-0.5)]
\end{align*}
\]

• A revolving balls and pentagons:

\[
\begin{align*}
\text{revolveRegion} &= \text{tx} (\sin \text{time}, \cos \text{time}) \\
\text{revBallB} &= \text{revolveRegion ballB} \\
\text{revPentaB} &= \text{revolveRegion pentaB}
\end{align*}
\]
Power Tools: Conditional Behaviors

• We can really start building a whole new language when we start adding conditional behaviors:

```haskell
cond :: Behavior Bool -> Behavior a -> Behavior a -> Behavior a
cond = lift3 $ \b x y -> if b then x else y
```

• Behavioral comparisons:

```haskell
(>*)) = lift2 (>)
(<*)) = lift2 (<)
```

• Alternating behaviors:

```haskell
flash = cond (cos time >* 0) red yellow
flash' = cond (cos time >* 0) green blue
```
Power Tools: Domain-Specific Type Classes

• Are there operations that apply to several different abstractions within our DSL?

• What about the concept of “over” – one shape, region, picture or behavior “over” top of another?

```haskell
class Combine a where
  empty :: a
  over :: a -> a -> a
```

• Write functions to layer all elements of a list:

```haskell
overMany :: Combine a => [a] -> a
overMany = foldr over empty
```
Power Tools: Domain-Specific Type Classes

class Combine a where
  empty :: a
  over :: a -> a -> a

• Write instances of the new class for pictures and behaviors

  instance Combine Picture where
    empty = EmptyPic
    over = Over

  instance Combine a => Combine (Behavior a) where
    empty = lift0 empty
    over = lift2 over
class Combine a where
  empty :: a
  over  :: a -> a -> a

instance Combine Picture where ...

instance Combine a => Combine (Behavior a) where ...

- Play with the new type classes:

overMany = foldr over empty

anim5 = animateB "Many Spheres" $ overMany [b1,b2,b3]
  where b1 = reg flash $ tx ((sin time)-1, cos time) ballB
  b2 = reg flash' $ tx ((sin time)+1, cos time) ballB
  b3 = reg flash'' $ tx (2 * sin time, cos time) pentaB
More Demos

- Check out the use of conditional animations and new type classes in these programs:
  
  anim2

  anim3

  anim4

  ...

  anim9

- Read through the rest of the animation notes
SUMMARY!
Summary

• Defining a new embedded DSL involves
  – defining **key abstract types** to be used by the client programs
  – defining **reuseable operations** over those abstract types

• Along the way, we saw:
  – heavy use of **functions as data**
  – the idea of **lifting** a Haskell function to a new abstract domain
  – the use of **type classes**
    • new instances for existing classes: related operations on new types
    • new classes: new domain-specific operations

• Historical note: Programming language researchers from 90s onward spent years defining and refining the basic principles of DSL design and looking for the right reusable, modular abstractions. And the research continues. Moreover, getting the specifics right is a fun, ongoing challenge in many domains.