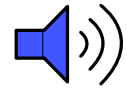
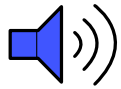
Lecture 16: Embedded Domain Specific Language

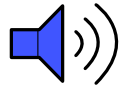


Outline:

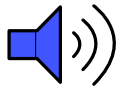
- Motivation
- Render pixels in ASCI
- Shape DSL

Motivation of EDSL 

- Semantic gap between computation domains and programming language abstractions.
- Shape domain:
  - Shapes are constructed as circles, squares.
  - Shapes are composed by operations such as intersect, invert, transform, translate.
  - Shapes are rendered as pixels on screen.
- Language domain:
  - Language provides types, expressions, and functions.
  - Language provides no shapes or pixels.
  - Application library is not strict and semantics is vague.

Design of EDSL 

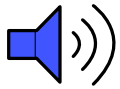
- Identify the primary domain abstractions
- Define constructors for primitive components
- Define operators to compose higher-level components
- Implement run function
  - interpreter function to *run* the abstraction
  - compiler function to *compile* the abstraction into lower-level representation.

Shape DSL 

- Shape abstraction[1]

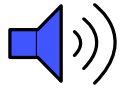
```haskell
data Shape =
             -- primitive shapes
             Empty | Disc | Square
             -- shape operators
             | Translate Vec Shape
             | Transform Matrix Shape
             | Union Shape Shape
             | Intersect Shape Shape
             | Invert Shape

data Vec     = Vec { vecX, vecY :: Double } -- for offset
data Matrix = Mat Vec Vec -- for rotation/deformation

-- run function: is a point in the shape
inside :: Point -> Shape -> Bool
```

[1] adopted from https://bitbucket.org/russo/afp-code/src/HEAD/L2/src/?at=master

Render shape in ASCI 

- Convert a shape into a string

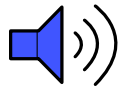
```
1  render :: Window  -- text window to display the shape
2         -> Shape   -- shape to be rendered
3         -> String  -- resulting string for the shape
4
5  render win sh = unlines -- [str] to str separated by \n
6                  $ map (concatMap putPixel) -- points to strings
7                        (pixels win) -- a list of a list of points
8    where
9      putPixel p  | p `inside` sh = "[]"
10                 | otherwise     = "  "
```

Display window 

- Define an abstract window for displaying rendered shapes

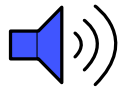
```
1  type Point = Vec    -- a point has x/y coordinates
2
3  data Window = Window
4    {  bottomLeft   :: Point    -- bottom left corner
5    ,  topRight     :: Point    -- upper right corner
6    ,  resolution   :: (Int, Int)  -- points per dimension
7    }
8
9  defaultWindow = Window  -- convenient constant
10   { bottomLeft   = point (-1.5) (-1.5)
11   , topRight     = point 1.5 1.5
12   , resolution   = (10, 10)
13   }
```

Display window



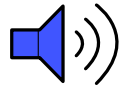
- Define function to convert window to points

```haskell
-- Generate [[Point]] corresponding to the pixels of a window
pixels :: Window -> [[Point]]
pixels (Window p0 p1 (w,h)) =
  [
     [
       -- points per row
       Vec x y | x <- samples (vecX p0) (vecX p1) w
     ]
       -- point rows from top to bottomw
       | y <- reverse $ samples (vecY p0) (vecY p1) h
  ]

-- Generate n evenly spaced numbers between x0 and x1
samples :: Double -> Double -> Int -> [Double]
samples x0 x1 n = take n
                  $ iterate (+dx) x0 -- increment by dx
  where
    dx = (x1 - x0) / fromIntegral (n - 1)
```

Run shape 

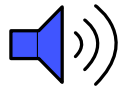
- A shape abstract is run by checking whether a point is inside the shape.

```
1  inside :: Point -> Shape -> Bool
2  _ `inside` Empty            = False -- no point in empty shape
3  p `inside` Disc             = distance p <= 1 -- disk shape
4  p `inside` Square           = maxnorm  p <= 1 -- square shape
5  -- move shape by vector 'v'
6  p `inside` Translate v s    = (p `sub` v) `inside` s
7  -- transform shape by matrix 'm'
8  p `inside` Transform m s    = (inv m `mul` p) `inside` s
9  p `inside` Union s1 s2      = p `inside` s1 || p `inside` s2
10 p `inside` Intersect s1 s2 = p `inside` s1 && p `inside` s2
11 p `inside` Invert s         = not (p `inside` s)
12
13 -- helper functions
14 distance :: Point -> Double
15 distance p = sqrt $ x*x + y*y
16   where x = vecX p; y = vecY p
17
18 maxnorm :: Point -> Double
19 maxnorm p = max (abs x) (abs y)
20   where x = vecX p; y = vecY p
```

Vector and matrix 

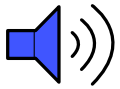
- Basic operations on vector and matrix

```haskell
data Vec    = Vec { vecX, vecY :: Double }
data Matrix = Mat Vec Vec

matrix :: Double -> Double -> Double -> Double -> Matrix
matrix a b c d = Mat (Vec a b) (Vec c d)

cross (Vec a b) (Vec c d) = a * c + b * d

-- Matrix multiplication
mul :: Matrix -> Vec -> Vec
mul (Mat r1 r2) v = Vec (cross r1 v) (cross r2 v)

-- Matrix inversion
inv :: Matrix -> Matrix
inv (Mat (Vec a b) (Vec c d)) =
                  matrix (d/k) (-b/k) (-c/k) (a/k)
  where k = a * d - b * c

-- Subtraction
sub :: Vec -> Vec -> Vec
sub (Vec x y) (Vec dx dy) = Vec (x - dx) (y - dy)
```

Render shape 

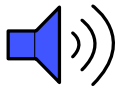
- Render an inverted disc

```
run :: Shape -> IO()
run shape = putStrLn $ render defaultWindow shape

run $ invert disc

[][][][][][][][][][]
[][][][][][][][][][]
[][][]          [][][]
[][]              [][]
[][]              [][]
[][]              [][]
[][]              [][]
[][][]          [][][]
[][][][][][][][][][]
[][][][][][][][][][]
```

Render shape 

- Render an inverted disc that is zoomed in 50%

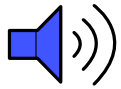
```
1  run :: Shape -> IO()
2  run shape = putStrLn $ render defaultWindow shape
3
4  zoom_in n shape = Transform m shape
5      where m = matrix (n/100+1) 0
6                       0         (n/100+1)
7
8  run $ zoom_in 50 $ invert disc
9
10 [][][][][][][][][][]
11 [][]               [][]
12 []                   []
13 []                   []
14 []                   []
15 []                   []
16 []                   []
17 []                   []
18 [][]               [][]
19 [][][][][][][][][][]
```

Render shape 

- Render a square

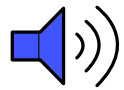
```
1  run :: Shape -> IO()
2  run shape = putStrLn $ render defaultWindow shape
3
4  run $ square
5
6
7
8        [] [] [] [] [] []
9        [] [] [] [] [] []
10       [] [] [] [] [] []
11       [] [] [] [] [] []
12       [] [] [] [] [] []
13       [] [] [] [] [] []
```

Render shape



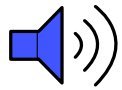
- Render a rotated square

```
1  run :: Shape -> IO()
2  run shape = putStrLn $ render defaultWindow shape
3
4  -- rotate clock-wise by degrees
5  rotate degree s = Transform m s
6      where m = matrix (cos alpha) (-(sin alpha))
7                       (sin alpha) (cos alpha)
8            alpha = pi * (degree/180)
9
10 run $ rotate 30 square -- rotate 30 degrees
11
12           []
13         [] [] [] []
14       [] [] [] [] [] []
15       [] [] [] [] [] [] []
16     [] [] [] [] [] [] []
17       [] [] [] [] [] []
18         [] [] [] []
19             []
```

Render shape 

- Render a rotated and scaled square

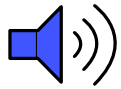
```
1  run :: Shape -> IO()
2  run shape = putStrLn $ render defaultWindow shape
3
4  scale :: Double -> Shape -> Shape -- scale a shape by 's'
5  scale s = Transform (matrix s  0
6                              0  s)
7
8  -- scale up by 1.2 and then rotate 30 degrees clockwise
9  run $ rotate 30 $ scale 1.2 square
10        []
11        [][][]
12      [][][][][][]
13      [][][][][][][][]
14    [][][][][][][][]
15    [][][][][][][][]
16  [][][][][][][][]
17      [][][][][][]
18          [][][]
19              []
```

Render shape 

- Render a stretched shape

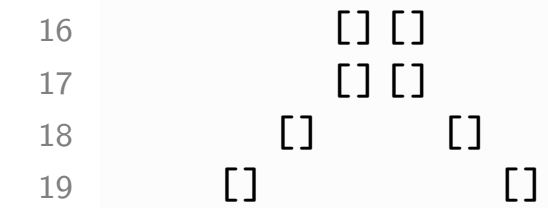
```
1  run :: Shape -> IO()
2  run shape = putStrLn $ render defaultWindow shape
3
4  -- stretch a shape by X-axis
5  stretchX s = Transform (matrix s 0
6                                  0 1)
7
8  -- stretch a square by X-axis to make a stick
9  stick d = rotate d $ stretchX 0.1 square
10
11 run $ stick 45  -- a stick rotated 45 degrees
12
13
14
15              []
16            []
17          []
18        []
```

Render shape



- Render a unioned shape

```
run :: Shape -> IO()
run shape = putStrLn $ render defaultWindow shape

stick d = rotate d $ stretchX 0.1 square
-- shift to the left
left = Translate (vec (-0.25) 0) $ stick 45
-- shift to the right
right = Translate (vec (0.25) 0) $ stick (-45)
-- union two shapes
arms = left `union` right

run arms
```



Render shape

- Render a unioned shape

```
stick d = rotate d $ stretchX 0.1 square
-- shift to the left
left = Translate (vec (-0.25) 0) $ stick 45
-- shift to the right
right = Translate (vec (0.25) 0) $ stick (-45)
-- union two shapes
arms = left `union` right
-- shift downwards
legs = Translate (vec 0 (-1)) arms
-- scale a square to make a small block
block = Translate (vec 0 1) (scale 0.6 square)

run $ arms `union` legs `union` block
```

```
      [][][][]
      [][][][]
      [][][][]
      [][][][]
        [][]
      []    []
    []  [][]  []
        [][]
      []    []
    []        []
```