AFP - Lecture 2
Domain Specific Embedded Languages

Patrik Jansson

2014

(slides by Norell, Bernardy & Jansson)

Anatomy of a DSEL

- A set of types modeling concepts in the domain
- *Constructor functions* constructing elements of these types
- *Combinators* combining and modifying elements
- *Run functions* making observations of the elements

newtype Signal a = Signal (Time -> a)

constS :: a -> Signal a
timeS :: Signal Time

($$) :: Signal (a -> b) -> Signal a -> Signal b
mapS :: (a -> b) -> Signal a -> Signal b

sample :: Signal a -> (Time -> a)

Primitive and Derived operations

- A *primitive operation* is defined exploiting the definitions of the involved types
- A *derived operation* can be defined purely in terms of other operations

timeS :: Signal Time
timeS = Signal (\t -> t)

mapS :: (a -> b) -> Signal a -> Signal b
mapS f s = constS f $$ s

Try to keep the set of primitive operations as small as possible! (Why?)

Think about

- Compositionality
  - Combining elements into more complex ones should be easy and natural
- Abstraction
  - The user shouldn't have to know (or be allowed to exploit) the underlying implementation of your types

Answer: Awkwardly!
addS x y = mapS (\t -> sample x t + sample y t) timeS

Suppose we didn't have ($$) in our Signal language. How would you define addS x y = constS (+) $$ x $$ y

Changing implementation shouldn't break user code!

Implementation of a DSEL

- Shallow embedding
  - Represent elements by their semantics (what observations they support)
  - Constructor functions and combinators do most of the work, run functions for free
- Deep embedding
  - Represent elements by how they are constructed
  - Most of the work done by the run functions, constructor functions and combinators for free
- Or something in between...

Is the signal library a deep or shallow embedding?

A deep embedding of Signals

```
data Signal a where
  ConstS    :: a -> Signal a
  TimeS     :: Signal Time
  (:$$)     :: Signal (a -> b) -> Signal a -> Signal b

constS   = ConstS
timeS    = TimeS
($$)     = (:$$)

sample :: Signal a -> (Time -> a)
sample (ConstS x)   = const  x
sample TimeS        = id
sample (f :$$ x)    = \t -> sample f t  $  sample x t

-- Start of derived operations
mapS :: (a -> b) -> Signal a -> Signal b
mapS f x = constS f $$ x
```

Generalized Algebraic Datatype (GADT). More on these in another lecture.

Simple constructors and combinators.

All the work happens in the run function.

Derived operations are unaffected by implementation style.

Deep vs. Shallow

- A shallow embedding (when it works out) is often more elegant
  - When there is an obvious semantics, embeddings usually work out nicely
- A deep embedding is easier to extend
  - Adding new operations
  - Adding new run functions
  - Adding optimizations

Like in the Signal example

Working out the type might be very difficult...

Most of the time you get a mix between deep and shallow!

Deep embedding may give you an easier start

More on this in another lecture.

Case Study: A language for Shapes

- Step 1: Design the interface

```
type Shape
-- Constructor functions
empty    :: Shape
disc     :: Shape
square   :: Shape
-- Combinators
translate    :: Vec ->   Shape -> Shape
scale        :: Vec ->   Shape -> Shape
rotate       :: Angle -> Shape -> Shape
union        :: Shape -> Shape -> Shape
intersect    :: Shape -> Shape -> Shape
difference   :: Shape -> Shape -> Shape
-- Run functions
inside   :: Point -> Shape -> Bool
```

Unit disc and unit square. Use translate and scale to get more interesting discs and rectangles.

Interface, continued

- Think about primitive/derived operations
  - No obvious derived operations
  - Sometimes introducing additional primitives makes the language nicer

```
invert    :: Shape -> Shape
transform   :: Matrix -> Shape -> Shape

scale :: Vec -> Shape -> Shape
scale v = transform (matrix (vecX v) 0 0 (vecY v))

rotate :: Angle -> Shape -> Shape
rotate a = transform (matrix     (cos a)  (-sin a)
                                 (sin a)  (cos a) )

difference :: Shape -> Shape -> Shape
difference a b = a `intersect` invert b
```

We need a language for working with matrices!

Do you remember your linear algebra course?

Side track: A matrix library

```
type Matrix
type Vector
type Point

-- Constructor functions
point      :: Double -> Double -> Point
vec        :: Double -> Double -> Vec
matrix     :: Double -> Double -> Double -> Double -> Matrix
-- Combinators
mulPt      :: Matrix -> Point -> Point
mulVec     :: Matrix -> Vec -> Vec
inv        :: Matrix -> Matrix
subtract   :: Point -> Vec -> Point
-- Run functions
ptX, ptY   :: Point -> Double
vecX, vecY  :: Vec -> Double
```

This should do for our purposes.

Shallow embedding

- What are the observations we can make of a shape?
  - inside :: Point -> Shape -> Bool
  - So, let's go for

```
newtype Shape = Shape (Point -> Bool)

inside :: Point -> Shape -> Bool
inside p (Shape f) = f p
```

In general, it's not this easy. In most cases you need to generalize the type of the run function a little to get a compositional shallow embedding.

Shallow embedding, cont.

- If we picked the right implementation the operations should now be easy to implement

```
empty       = Shape $ \p -> False
disc        = Shape $ \p -> ptX p ^ 2 + ptY p ^ 2 <= 1
square      = Shape $ \p -> abs (ptX p) <= 1 && abs (ptY p) <= 1

transform m a   = Shape $ \p -> mulPt (inv m) p `inside` a
translate v a   = Shape $ \p -> subtract p v `inside` a

union a b       = Shape $ \p -> inside p a || inside p b
intersect a b   = Shape $ \p -> inside p a && inside p b
invert a        = Shape $ \p -> not (inside p a)
```

Trick: move the point instead of the shape

Deep embedding

- Representation is easy, just make a datatype of the primitive operations

```
data Shape where -- using Gen. Alg. DataType syntax
  -- Constructor functions
  Empty      :: Shape
  Disc       :: Shape
  Square     :: Shape
  -- Combinators
  Translate  :: Vec ->     Shape -> Shape
  Transform :: Matrix -> Shape -> Shape
  Union      :: Shape -> Shape -> Shape
  Intersect  :: Shape -> Shape -> Shape
  Invert     :: Shape -> Shape

empty = Empty; disc = Disc; ...
```

Deep embedding

- … the same datatype without GADT notation:

```
data Shape =   Empty | Disc | Square
            |  Translate Vec Shape
            |  Transform Matrix Shape
            |  Union Shape Shape | Intersect Shape Shape
            |  Invert Shape

empty      = Empty
disc       = Disc
translate  = Translate
transform = Transform
union      = Union
intersect  = Intersect
invert     = Invert
```

Deep embedding, cont.

- All the work happens in the run function:

```
inside :: Point -> Shape       -> Bool
p `inside` Empty               = False
p `inside` Disc                = ptX p ^ 2 + ptY p ^ 2    <= 1
p `inside` Square              = abs (ptX p) <= 1 && abs (ptY p) <= 1
p `inside` Translate v a       = subtract p v `inside` a
p `inside` Transform m a       = mulPt (inv m) p `inside` a
p `inside` Union a b           = inside p a  ||   inside p b
p `inside` Intersect a b       = inside p a && inside p b
p `inside` Invert a            = not (inside p a)
```

Abstraction!

```
module Shape
    ( module Matrix
    , Shape
    , empty, disc, square
    , translate, transform, scale, rotate
    , union, intersect, difference, invert
    , inside
    ) where

import Matrix
...
```

It might be nice to re-export the matrix library

Hide the implementation of the Shape datatype

The interface is the same for both deep and shallow embedding. No visible difference to the user!

More interesting run function: render to ASCII-art

```
module Render where

import Shape

data Window = Window
     {bottomLeft      :: Point
     , topRight        :: Point
     , resolution       :: (Int, Int)
     }

defaultWindow :: Window
pixels    :: Window -> [[Point]]

render :: Window -> Shape -> String
render win a = unlines $ map (concatMap putPixel) (pixels win)
     where
          putPixel p   | p `inside` a     = “[]”
                       | otherwise        = “  ”
```

Some action

```
module Animate where

import Shape
import Render
import Signal

animate :: Window -> Time -> Time -> Signal Shape -> IO ()
```

- Go live!

Discussion

- Adding coloured shapes
  - Go back and discuss what changes would need to be made
- Bad shallow implementations
  - Looking at the render run function we might decide to go for

    ```
    newtype Shape = Shape (Window -> String)
    ```
  - Discuss the problems with this implementation
- Other questions/comments..?

Summary

- Difierent kinds ofioperations
  - constructor functions / combinators / run functions
  - primitive / derived
- Implementation styles
  - Shallow – representation given by semantics
  - Deep – representation given by operations
- Remember
  - Compositionality
  - Abstraction