## Lecture 2 Domain Specific Embedded Languages

Patrik Jansson 2011

(slides by Norell and Bernardy)

01/18/11

## Anatomy of a DSEL

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

01/18/11

## Primitive and Derived operations

- A primitive operation is defined exploiting the definitions of the involved types timeS :: Signal Time timeS = Signal (\t -> t)
- A derived operation can be defined purery in terms of other operations

### Think ahout

- 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 tyr

01/18/11

## 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...

01/18/11

data Signal a where :: a -> Signal a :: Signal Time :: Signal (a -> b) -> Signal a -> Signal b

A deep embedding of Signals

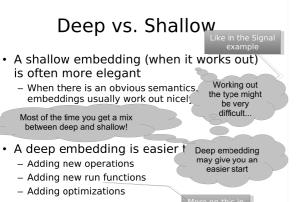
ConstS TimeS (:\$\$)

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

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

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

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

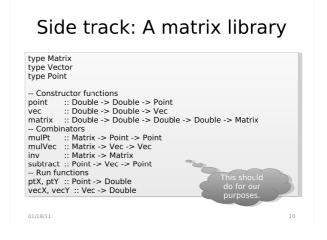


## Shapes • Step 1: Design the interface type Shape -- Constructor functions empty :: Shape circle :: Shape square :: Shape -- Combinators translate :: Vec -> Shape -> Shape scale :: Vec -> Shape -> Shape rotate :: Angle -> Shape -> Shape intersect :: Shape -> Shape -> Shape difference :: Shape -> Shape -> Shape -- Run functions inside :: Point -> Shape -> Bool

Case Study: A language for

## 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 :: Vec -> Shape -> Shape scale :: Angle -> Shape -> Shape difference :: Shape -> Shape -> Shape difference :: Shape -> Shape -> Shape difference :: Shape -> Shape -> Shape difference a b = a `intersect` invert b

01/18/13



# 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

```
Shallow embedding, cont.

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

empty = Shape $\p -> False \\
circle = Shape $\p -> ptX p ^2 + ptY p ^2 <= 1 \\
square = Shape $\p -> ptX p ^2 + ptY p ^2 <= 1 \\
square = Shape $\p -> mulPt (inv m) p 'inside' a \\
transform m a = Shape $\p -> subtract p v 'inside' a \\
union a b = Shape $\p -> inside p a || inside p b \\
intersect a b = Shape $\p -> not (inside p a)
```

## Deep embedding

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

```
data Shape where
-- Constructor functions
Empty :: Shape
Circle :: 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; circle = Circle; ...
```

## Deep embedding

• ... the same datatype without GADT notation:

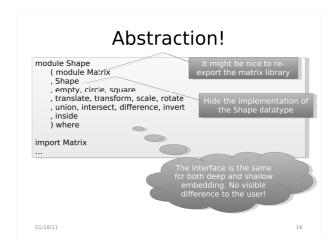
```
data Shape = Empty | Circle | Square | Translate Vec Shape | Transform Matrix Shape | Union Shape Shape | Intersect Shape Shape | Invert Shape | Intersect Shape Shape | Invert Shape | Intersect Shape Shape | Invert Shape | Intersect Shape Shape | Invert Shape | Intersect Shape Shape Shape | Intersect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

## Deep embedding, cont.

• All the work happens in the run function:

```
inside :: Point -> Shape
p `inside` Empty
p `inside` Circle
p `inside` Circle
p `inside` Translate v a
p `inside` Transform m a
p `inside` Union a b
p `inside` intersect a b
p `inside` Invert a

-> Bool
= ptX p ^2 + ptY p ^2 <= 1
= abs (ptX p) <= 1 && abs (ptY p) <= 1
= abs (ptX p) <= 1 && abs (ptY p) <= 1
= abs (ptX p) <= 1 && abs (ptY p) <= 1
= inside p a || inside p b
= inside p a && inside p b
= not (inside p a)
```



## 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 = ""
```

## module Animate where import Shape import Render import Signal animate :: Window -> Time -> Signal Shape -> IO () • Go live!

01/18/11

## 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

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

01/19/11

20