EECS 776
Functional Programming

Dr. Andy Gill

University of Kansas

November 20th, 2015

KU

Domain Specific Languages

Based on slides by Ulf Norell. I emailed him asking permission:

> Ulf,
> I'm teaching an AFP class here at Kansas, and really like the way you've presented the concept of a DSL. Can I based my slides of your material?

He replied:

KU

Domain Specific Languages

Based on slides by Ulf Norell. I emailed him asking permission:

> Ulf,
> I'm teaching an AFP class here at Kansas, and really like the way you've presented the concept of a DSL. Can I based my slides of your material?

He replied:

> Du har skickat ett brev till en epostadress som r under avveckling, vnligen se webbsidan nedan eller kontakta mottagaren fr att f rtt adress.
> Ditt brev har levererats till mottagaren.

KU

Domain Specific Languages

Based on slides by Ulf Norell. I emailed him asking permission:

> Ulf,
> I'm teaching an AFP class here at Kansas, and really like the way you've presented the concept of a DSL. Can I based my slides of your material?

He replied:

> Du har skickat ett brev till en epostadress som r under avveckling, vnligen se webbsidan nedan eller kontakta mottagaren fr att f rtt adress.
> Ditt brev har levererats till mottagaren.

> You have sent an email to an email address which is being phased out, please see the webpage below or contact the recipient to get the correct address.
> Your email has been delivered to the recipient.

Domain Specific Languages

- We have seen several DSLs.
- All are libraries in Haskell.
- You need to know Haskell to use any Haskell-hosted DSL.

KU

Domain Specific Languages - Anatomy

- User-code, written in the DSL are centered round a specific type (or types).
- There are ways of
  - constructing this type,
  - composing this type,
  - running (or making observations about) this type.

KU

What to think about

- Compositionally

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.

(or changing implementation shouldn't break user code!)

KU

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 function(s) are (almost) for free
- Deep embedding
  - Represent elements by how they are constructed
  - Most of the work done by the run function(s)
  - Constructor functions and combinators for free
  - Optimization opportunities (more later)

KU

```
-- our principal type, Expr
newtype Expr = Expr (Maybe Int)
```

```
-- our principal type, Expr
newtype Expr = Expr (Maybe Int)

-- our way of constructing Expr's
lit :: Int -> Expr
lit n = Expr (Just n)
```

```
-- our principal type, Expr
newtype Expr = Expr (Maybe Int)

-- our way of constructing Expr's
lit :: Int -> Expr
lit n = Expr (Just n)

-- our way of composing Expr's
plus :: Expr -> Expr -> Expr
plus (Expr (Just v1)) (Expr (Just v2))
                    = Expr (Just (v1 + v2))
plus _ _ = Expr Nothing
```

```
-- our principal type, Expr
newtype Expr = Expr (Maybe Int)

-- our way of constructing Expr's
lit :: Int -> Expr
lit n = Expr (Just n)

-- our way of composing Expr's
plus :: Expr -> Expr -> Expr
plus (Expr (Just v1)) (Expr (Just v2))
                    = Expr (Just (v1 + v2))
plus _ _ = Expr Nothing

-- our way of running Expr's
runExpr :: Expr -> Maybe Int
runExpr (Expr v) = v
```

KU

```
divide :: Expr -> Expr -> Expr
divide (Expr (Just v1)) (Expr (Just v2))
       | v2 /= 0 = Expr (Just (v1 `div` v2))
divide _ _ = Expr Nothing
```

```
*Main> runExpr (lit 1)
```

```
divide :: Expr -> Expr -> Expr
divide (Expr (Just v1)) (Expr (Just v2))
       | v2 /= 0 = Expr (Just (v1 `div` v2))
divide _ _ = Expr Nothing
```

```
*Main> runExpr (lit 1)
Just 1
*Main> runExpr (lit 1 `plus` lit 2)
```

```
divide :: Expr -> Expr -> Expr
divide (Expr (Just v1)) (Expr (Just v2))
       | v2 /= 0 = Expr (Just (v1 `div` v2))
divide _ _ = Expr Nothing
```

```
*Main> runExpr (lit 1)
Just 1
*Main> runExpr (lit 1 `plus` lit 2)
Just 3
*Main> runExpr (lit 1 `divide` lit 2)
```

```
divide :: Expr -> Expr -> Expr
divide (Expr (Just v1)) (Expr (Just v2))
       | v2 /= 0 = Expr (Just (v1 `div` v2))
divide _ _ = Expr Nothing
```

```
*Main> runExpr (lit 1)
Just 1
*Main> runExpr (lit 1 `plus` lit 2)
Just 3
*Main> runExpr (lit 1 `divide` lit 2)
Just 0
*Main> runExpr (lit 1 `divide` lit 0)
```

```
divide :: Expr -> Expr -> Expr
divide (Expr (Just v1)) (Expr (Just v2))
       | v2 /= 0 = Expr (Just (v1 `div` v2))
divide _ _ = Expr Nothing
```

```
*Main> runExpr (lit 1)
Just 1
*Main> runExpr (lit 1 `plus` lit 2)
Just 3
*Main> runExpr (lit 1 `divide` lit 2)
Just 0
*Main> runExpr (lit 1 `divide` lit 0)
Nothing
```

KU

This is an API, implementation can be hidden.

```
-- our principal type, Expr
newtype Expr = ...
```

This is an API, implementation can be hidden.

```
-- our principal type, Expr
newtype Expr = ...

-- our way of constructing Expr's
lit :: Int -> Expr
```

This is an API, implementation can be hidden.

```
-- our principal type, Expr
newtype Expr = ...

-- our way of constructing Expr's
lit :: Int -> Expr

-- our way of composing Expr's
plus :: Expr -> Expr -> Expr
divide :: Expr -> Expr -> Expr
```

This is an API, implementation can be hidden.

```
-- our principal type, Expr
newtype Expr = ...

-- our way of constructing Expr's
lit :: Int -> Expr

-- our way of composing Expr's
plus :: Expr -> Expr -> Expr
divide :: Expr -> Expr -> Expr

-- our way of running Expr's
runExpr :: Expr -> Maybe Int
```

Deep Embedding

```
-- our principal type, Expr
data Expr = Lit Int
          | Plus Expr Expr
          | Divide Expr Expr
```

Deep Embedding

```
-- our principal type, Expr
data Expr = Lit Int
          | Plus Expr Expr
          | Divide Expr Expr

-- our way of constructing Expr's
lit :: Int -> Expr
lit n = Lit n
```

Deep Embedding

```
-- our principal type, Expr
data Expr = Lit Int
          | Plus Expr Expr
          | Divide Expr Expr

-- our way of constructing Expr's
lit :: Int -> Expr
lit n = Lit n

-- our way of composing Expr's
plus :: Expr -> Expr -> Expr
plus e1 e2 = Plus e1 e2

divide :: Expr -> Expr -> Expr
divide e1 e2 = Divide e1 e2
```

KU

```
runExpr :: Expr -> Maybe Int
runExpr (Lit i) = Just i
runExpr (Plus e1 e2) =
  case runExpr e1 of
    Nothing -> Nothing
    Just v1 -> case runExpr e2 of
                 Nothing -> Nothing
                 Just v2 -> Just (v1 + v2)
runExpr (Divide e1 e2) =
  case runExpr e1 of
    Nothing -> Nothing
    Just v1 -> case runExpr e2 of
                 Nothing -> Nothing
                 Just v2 | v2 == 0    -> Nothing
                         | otherwise -> Just (v1 `div` v2)
```

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 function(s) are (almost) for free
- Deep embedding
  - Represent elements by how they are constructed
  - Most of the work done by the run function(s)
  - Constructor functions and combinators for free
  - Optimization opportunities (more later)

KU

A Language for Shapes

Step 1: Design the interface (or API)

```
data Shape
-- Constructor functions
empty    :: Shape
circle   :: Shape  -- unit circle
square   :: Shape  -- unit square
-- 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
```

KU

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
```

Shallow embedding

What are the observations we can make of a shape?

- `inside :: Point -> Shape -> Bool`
- So, lets go for

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

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

In general, its not this easy. In most cases you need to generalize the type of the internal function a little to get a compositional shallow embedding.

KU

Shallow embedding, continued

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 -> 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)
```

KU

Deep embedding

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

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

empty           = Empty
circle          = Circle
translate       = Translate
transform       = Transform
union           = Union
intersect       = Intersect
invert          = Invert
```

Deep embedding, continued

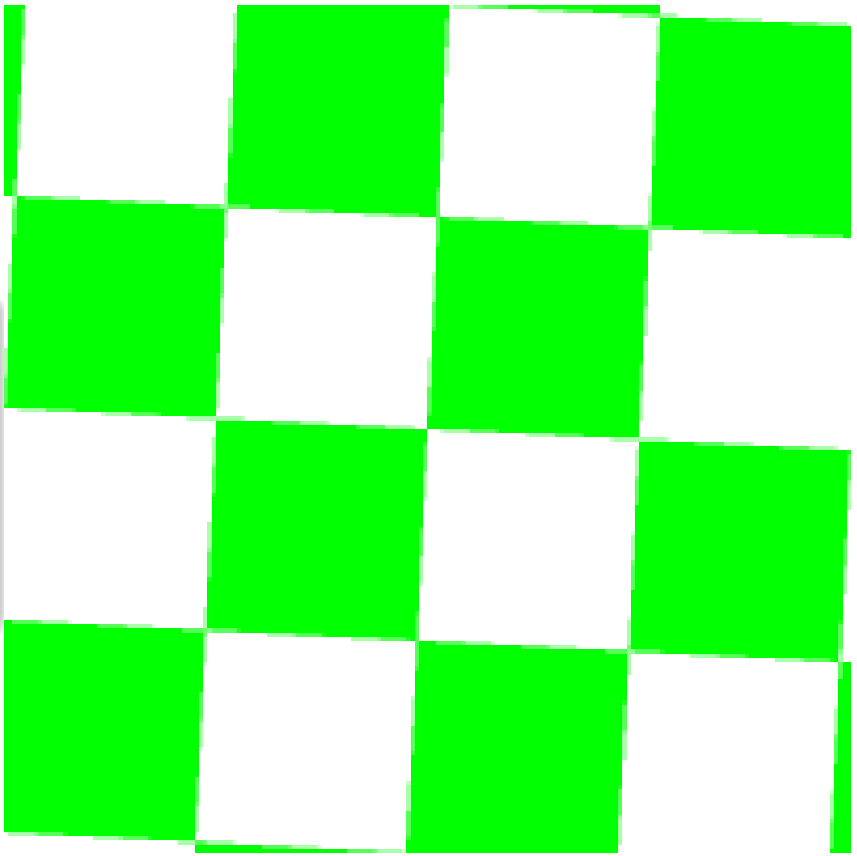
All the work happens in the run function.

```
inside :: Point -> Shape -> Bool
p `inside` Empty          = False
p `inside` Circle         = ptX p ^ 2 + ptY p ^ 2 <= 1
p `inside` Square         = abs (ptX p) <= 1 && abs (ptY p) <=
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)
```

KU

Example DSL: Chalkboard

```
...
$ fmap (choose green white)
$ rotate 0.05
$ scale 50
$ checker
```



KU

Example DSL: Chalkboard

```
...
$ stack
$ [ fmap (choose (withAlpha 0.8 col) (transparent white))
     $ functionline (\ x -> (x * 400,t + 80 * x * sin (x * t))) 3 count
  | (col,t) <- zip [red,green,blue] [16,18,20]
  ] ++
  [ fmap (choose (withAlpha 0.8 col) (transparent white))
     $ functionline (\ x -> (400 - x * 400,t - 80 * x * sin (x * t))) 3 count
  | (col,t) <- zip [cyan,purple,yellow] [15,17,20]
  ] ++
  [ pure (alpha white) ]
```