Lab 11 - Intro Embedded Domain Specific Language

CS 1XA3

March 27^{th}, 2018

Recap: Last Lab

- Recall our expression type from last lab

```haskell
data BExpr a = And (BExpr a) (BExpr a)
             | Or  (BExpr a) (BExpr a)
             | Not (BExpr a)
             | Const a
             | Var String
```

- We wrote an evluation function for it

```haskell
eval :: Map.Map String Bool -> BExpr Bool -> Bool
```

- What if we wanted to generalize our evaluation to

```haskell
eval :: Map.Map String b -> BExpr b -> b
```

Domain Specific Language

We can generalize our expressions into a Embedded Domain Specific Language using a Type Class

```haskell
class (Eq a) => BoolAlgebra a where
  {- Class Methods (Required Implementation) -}
  bTrue    :: BExpr a
  bFalse   :: BExpr a
  bEval    :: Map.Map String a -> BExpr a -> a

  {- Methods with Default Implementations -}
  bAnd    :: BExpr a -> BExpr a -> BExpr a
  bAnd    = And
  bOr     :: BExpr a -> BExpr a -> BExpr a
  bOr     = Or
  bNot    :: BExpr a -> BExpr a
  bNot    = Not
```

Domain Specific Language

We can provide a Bool instance for working with True and False

```
instance BoolAlgebra Bool where
  bTrue = Const True
  bFalse = Const False
  bEval vrs expr = case expr of
    (And e1 e2) -> (bEval vrs e1) && (bEval vrs e2)
    (Or e1 e2)  -> (bEval vrs e1) || (bEval vrs e2)
    (Not e)     -> not $ bEval vrs e
    (Const x)   -> x
    (Var nm)    -> case Map.lookup nm vrs of
      (Just val) -> val
      Nothing -> error "Error: eval failed lookup"
```

Domain Specific Language

- Now if we want to encode an expression, instead of writing

```
expr :: BExpr Bool
expr = And (Const False)
           (Or (Const True) (Const False))
```

- We can generalize with

```
expr :: BoolAlgebra a => BExpr a
expr = bFalse `bAnd` (bTrue `bOr` bFalse)

ans :: Bool
ans = bEval (Map.fromList []) expr
```

SuperCharging Our DSL

Recall our cnf function from the last lab that rewrites BExpr into Conjuctive Normal Form

```haskell
cnf :: BExpr a -> BExpr a
-- Double Negation
cnf (Not (Not e))         = cnf e
-- De Morgans Laws
cnf (Not (Or e1 e2))      = cnf $ And (Not e1) (Not e2)
cnf (Not (And e1 e2))     = cnf $ Or (Not e1) (Not e2)
-- Distributivity
cnf (Or e1 (And e2 e3)) = cnf $ And (Or e1 e2) (Or e1 e3)
cnf (Or (And e1 e2) e3) = cnf $ And (Or e1 e3) (Or e2 e3)
```

SuperCharging Our DSL

- Let's alter our DSL implementation a little to make use of our cnf rewrite by default

```
class Eq a => BoolAlgebra a where
  {- Methods with Default Implementations -}
  bAnd    :: BExpr a -> BExpr a -> BExpr a
  bAnd e1 e2  = cnf $ And e1 e2
  bOr     :: BExpr a -> BExpr a -> BExpr a
  bOr  e1 e2  = cnf $ Or e1 e2
  bNot    :: BExpr a -> BExpr a
  bNot e  = cnf $ Not e
```

- Now any expression written in the DSL is automatically rewritten to Conjunctive Normal Form!