# GADTs

## For Eliminating Runtime Checks

Vijay Anant

Lambda Matters

November 24, 2018

# Algebraic Data Types

```
data Point = Pt Int Int
data Expr  = Number Integer | Boolean Bool
```

```
ghci> let a = Number 10
ghci> let b = Boolean True
ghci> :t a
a :: Expr
ghci> :t b
b :: Expr
```

```
ghci> :t Number
Number :: Integer -> Expr
ghci> :t Boolean
Boolean :: Bool -> Expr
```

# Expression Evaluator

### Expression Type

```
data Expr = Lit Int
          | Succ Expr
          | IsZero Expr
          | If Expr Expr Expr
```

### The Value Type

```
data Value = IntVal Int | BoolVal Bool
```

# Expression Evaluator

The expression evaluator is a function that takes an *Expr* and returns a *Value*

```haskell
eval :: Expr -> Value
eval (Lit i)      = IntVal i
eval (Succ e)     = case eval e of
        IntVal  i -> IntVal (i+1)
eval (IsZero e)   = case eval e of
        IntVal  i -> BoolVal (i==0)
eval (If b e1 e2) = case eval b of
        BoolVal True  -> eval e1
        BoolVal False -> eval e2
```

# Invalid Expressions

Our *Expr* type allows some expressions that are not valid!

```
expr1 = Succ (Lit 1)                -- valid and type checks
expr2 = Succ (IsZero (Lit 1))       -- invalid but type checks
expr3 = If (Lit 0) (Lit 1) (Lit 2) -- invalid but type checks
```

When is an expression invalid?

- Valid mental model but doesn't type check
- Type checks but invalid mental model

# The Problem

**The eval function is partial**

```
eval :: Expr -> Value
eval (Lit i)       = IntVal i
eval (Succ e)      = case eval e of
    IntVal  i -> IntVal (i+1)
  -- BoolVal b -> ???
eval (IsZero e)    = case eval e of
    IntVal  i -> BoolVal (i==0)
  -- BoolVal b -> ???
eval (If b e1 e2) = case eval b of
    BoolVal True  -> eval e1
    BoolVal False -> eval e2
  -- IntVal  i     -> ???
```

# Possible Solutions

**What can we do now?**

- Expand the partial function to define a value for every point in the domain
- Restrict the domain to contain only those points for which the function is defined

**Note:**

- Expanding the function involves defining special value(s) (*error code*) for all the points where the function is not defined
- The error codes are outside the range of success values

# Generalised ADTs

- Generalizes ordinary data types
- Allow more compile time checks than ADTs
- Allow arbitrary return types for value constructors
- Type refinement when pattern matching
- GADTs are provided in GHC as a language extension

```
{-# LANGUAGE GADTs #-}

data Expr a where
  Number :: Int -> Expr Int
  Succ   :: Expr Int -> Expr Int
  IsZero :: Expr Int -> Expr Bool
  If     :: Expr Bool->Expr a->Expr a->Expr a
```

# Generalised ADTs

Now, invalid expressions are caught at compile time!

```
ghci> :t Succ (Lit 10)
Succ (Lit 10) :: Expr Int

ghci> :t Succ (IsZero (Lit 0))
<interactive>:1:7: error:
     Couldn't match type Bool with Int
      Expected type: Expr Int
        Actual type: Expr Bool
     In the first argument of Succ, namely (IsZero (Lit 0))
      In the expression: Succ (IsZero (Lit 0))
```

# Generalised ADTs

The *eval* function is now simple and *total* as we do not have to worry about invalid expressions

```
eval :: Expr a     -> a
eval (Number i)    =  i
eval (Succ e)      =  1 + eval e
eval (IsZero e)    =  0 == eval e
eval (If b e1 e2)  =  if eval b
                      then eval e1
                      else eval e2
```

# More on GADTs

- Type signature is needed for functions using GADTs
- You will end up needing *ScopedTypeVariables* at some point
- Dependent Types!

**Questions?**

https://vijayanant.github.io/