Statically type checked DSLs with GADTs

Konrad Werbliński

October 1, 2020

ADT

ADT

In their simplest form they are similar to enum in C++

Example

```
data OS = Windows | Linux | MacOS

enum class OS {Windows, Linux, MacOS};
```

ADT

Example

```
isGoodSystem os = case os of
  Windows -> False
  Linux -> True
  MacOS -> True

----------------
isGoodSystem Windows = False
isGoodSystem Linux = True
isGoodSystem MacOS = True
```

ADT

But each constructor can holds values! So they are similar to combination of variant and tuple, but each element of the alternative is named.

Example

```
data Shape
  = Circle Float Color
  | Rectangle Float Float Color
```

ADT

They can also be generic.

Example

```
data Optional a = Nothing | Just a

Just 5 :: Optional Int
Just "Konrad" :: Optional String
Nothing :: Optional a
```

ADT

They can also be recursive.

Example

```
data List a = Empty | Cons a (List a)

- In Haskell:

data [a] = [] | a : [a]

Usage:
5 : [1, 3, 4] = [5, 1, 3, 4]

head :: [a] -> a
head (x : _) = xs
```

ADT

They can also be recursive.

Example

```
data Tree a = Leaf | Node (Tree a) a (Tree a)

t :: Tree Int
t = Node Leaf 44 (Node Leaf 42 Leaf)

44
 \
  42
```

ADT

Let's look at the simple expressions DSL

Example

```
data Expr
  = ENum Int
  | EStr String
  | EPlus Expr Expr
  | ECat Expr Expr
  | ELen Expr

EPlus (ENum 44) (ELen (EStr "Munich"))
```

ADT

Let's try to evaluate expressions.

Example

```
data Value = VInt Int | VStr String

eval :: Expr -> Value
eval (EInt n) = VInt
...
eval (EPlus e1 e2) =
  let (VInt n1) = eval e1 in
  let (VInt n2) = eval e2 in
  VInt (n1 + n2)
...

But this will crash for:
EPlus (EStr "Munich") (EInt 44)
```

ADT

- We would like to make our DLS type safe!
- But how to do it?
- We can define expressions as Expr a, and use the type parameter to carry necessary information!

ADT

Example

```
data Expr a
  = ENum Int
  | EStr String
  | EPlus (Expr Int) (Expr Int)
  | ECat (Expr String) (Expr String)
  | ELen (Expr String)
```

However, above code would not work, we are not setting the type parameter of the created types.

ADT

We can wrapper functions over constructors, to set the type parameter.

Example

```
eNum :: Int -> Expr Int
eNum n = ENum n

ePlus :: Expr Int -> Expr Int -> Expr Int
ePlus e1 e2 = EPlus e1 e2

eLen :: Expr String -> Expr Int
eLen e = ELen e
```

ADT

However, we can do better! :D

GADT

GADT

We can use GADTs!

Example

```
data Expr a where
  ENum :: Int -> Expr Int
  EStr :: String -> Expr String
  EPlus :: Expr Int -> Expr Int -> Expr Int
  ECat :: Expr String -> Expr String -> Expr String
  ELen :: Expr String -> Expr Int
```

GADT

Now the following code will produce type error!

Example

```
EPlus (EStr "Munich") (EInt 44)
```

GADT

Eval function also becomes nicer!

Example

```
eval :: Expr a -> a
eval (EInt n) = n
...
eval (EPlus e1 e2) =
  let n1 = eval e1 in
  let n2 = eval e2 in
  n1 + n2
...
```

GADT - More complex DSL

Statically type checked printf

Example

```
data Format t where
  Str :: Format a -> Format (String -> a)
  Inr :: Format a -> Format (Int -> a)
  Flt :: Format a -> Format (Float -> a)
  Lit :: String -> Format a -> Format a
  Eol :: Format a -> Format a
  End :: Format ()

Lit "Hello" (Str (Lit "! Answer:" (Inr End)))
:: Format (String -> Int -> ())
```

GADT - More complex DSL

Statically type checked String formatting

Example

```
printf :: Format a -> a
printf End = ()
printf (Lit s format) = putStr s `seq` printf format
printf (Eol format) = putStrLn "" `seq` printf format
printf (Str format) =
  \x -> putStr x `seq` printf format
printf (Inr format) =
  \x -> (putStr . intToString) x `seq` printf formats
printf (Flt format) =
  \x -> (putStr . floatToString) x `seq` printf format
```

GADT - More complex DSL

Thus:

Example

```
Lit "Hello" (Str (Lit "! Answer:" (Inr End)))
:: Format (String -> Int -> ())

printf (Lit "Hello" (Str (Lit "! Answer:" (Inr End))))
:: String -> Int -> ()

printf
(Lit "Hello" . Str . Lit "! Answer:" . Inr $ End)
"Konrad" 42
```

GADT - empty types as labels

Let's imagine a tree that holds different types of data in the left and right sons.

Example

```
data Tree
  = Leaf
  | LeftSon Int Tree Tree
  | RightSon String Tree Tree

But nothing prevents us from creating:
Left (Left 42 Leaf Leaf) (Left 44 Leaf Leaf)
```

GADT - empty types as labels

How to make sure that the construction of a tree is correct. We can use GADTs!, and empty types for labeling.

Example

```
data Left
data Right

data Tree side where
  Leaf :: Tree a
  LeftSon ::
    Int -> Tree Left -> Tree Right -> Tree Left
  RightSon ::
    String -> Tree Left -> Tree Right -> Tree Right
```

GADT - encoding natural numbers as empty types

Example

```
data Zero
data Succ n

type One = Succ Zero
type Two = Succ One
//type Two = Succ (Succ Zero)
...

To make life easier we will write
0, 1, 2, 3 ... instead of types
Zero, One, Two, Three, ...
```

GADT - Vec

Example

```
data Vec n a where
  [] :: Vec 0 a
  (:) :: a -> Vec n a -> Vec (Succ n) a
```

GADT - Vec

Example

```
[42, 5, 44, 59] :: Vec 4 Int

["Haskell", "OCaml", "Bestrafer"] :: Vec 3 String

[] :: Vec 0 a
```

GADT - Vec

Example

```
map :: (t1 -> t2) -> Vec n t1 -> Vec n t2
map _ [] = []
map f (head : tail) = f head : map f tail
```

GADT - Vec

Example

```
map f [1, 2, 3, 4] = [f 1, f 2, f 3, f 4]
```

GADT - Vec

Example

```
map fac [1, 2, 3, 4] = [1, 2, 6, 24]
```

GADT - Vec

Example

```
map :: (t1 -> t2) -> Vec n t1 -> Vec n t2
map _ [] = []
map f (head : tail) = f head : map f tail
```

GADT - Vec

Example

```
map :: (t1 -> t2) -> List t1 -> List t2
map _ _ = []
```

GADT - Vec

Example

```
map :: (t1 -> t2) -> List t1 -> List t2
map f list = [f (head list)]
```

GADT - Vec

Example

```
map :: (t1 -> t2) -> Vec n t1 -> Vec n t2
map _ [] = []
map f (head : tail) = f head : map f tail
```

GADT - matrix operations

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ a_{31} & a_{32} & \cdots & a_{3m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nm} \end{bmatrix}.$$

```
Vec n (Vec m Int)
```

GADT - matrix operations

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ a_{31} & a_{32} & \cdots & a_{3m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nm} \end{bmatrix}.$$

```
Matrix n m
```

GADT - matrix operations

Example

```
transpose :: Matrix n m -> Matrix m n
transpose matrix =
  let indices = range (len (head matrix)) in
  map (\i -> column i matrix) indices
```

GADT - matrix operations

Example

```
mult :: Matrix n m -> Matrix m k -> Matrix n k
mult a b = map (\v -> multVec v b) a
```

GADT - matrix operations

Example

```
multVec :: Vec n Int -> Matrix n m -> Vec m Int
```

GADT - matrix operations

Example

```
mult :: Matrix n m -> Matrix m k -> Matrix n k
mult a b = map (\v -> multVec v b) a
```

GADT - red black tree

Example

```
data Black
data Red

data RBTree col blackHeight t where
  Black :: RBTree c1 n t ->
           t ->
           RBTree c2 n t ->
           RBTree Black (S n) t
  Red :: RBTree Black n t ->
         t ->
         RBTree Black n t ->
         RBTree Red n t
  Empty :: RBTree Black 0 t
```