# GADTs in Action

CIS 194 Week 11
8 April 2015

```
{-# LANGUAGE GADTs                                                              #-}
{-# LANGUAGE TypeFamilies                                                       #-}
{-# LANGUAGE TypeOperators                                                      #-}
```

Last week we starting learning about Generalized Algebraic Data Types or, as they are generally called, GADTs. Recall that GADTs are special data types whose constructors have a non-uniform return type. This allows us to encode invariants about a data structure in its type. We saw how GADTs can be used to encode the length of a vector. This allowed us to define vector addition in a "type-safe" way, enforcing the invariant that when you add two vectors their lengths must be equal in the type of the function.

## The Simple Typed Lambda Calculus

The Simply Typed Lambda Calculus, or STLC for short, is a basic functional programming language. It is the building block for modern day functional languages such as Haskell and OCaml, and is used extensively in programming language research. We will consider a variant of STLC with two base types: `Int` and `Bool`, as well as arrow types. Programs in STLC are composed of expressions. An expression is either an integer or boolean literal, a variable, an anonymous function (lambda abstraction), a function application, a binary operation, or an if statement. We can encode an abstract syntax tree for STLC using the following Haskell data type:

```
data Expr = IntLit Int
          | BoolLit Bool
          | Lambda String Expr
          | App Expr Expr
          | Bop Bop Expr Expr
          | If Expr Expr Expr

data Bop = Add | Sub | Eq | Lt | Gt | And | Or
```

This is a perfectly fine representation of the language, however it allows us to create expressions that are ill-typed. This means that in order to write an interpreter for this language, we would first need to write a type checker that verifies that the expressions we are interpreting are well-typed. It would be really great if we could encode the invariants about STLC's typing rules in to the expression data type.

### GADTs to the Rescue!

In fact, we *can* encode the STLC typing invariants in to the expression data type. To do this, we will have to use GADTs. Before we begin, we have to define a way to equate type-level representations of types (ie `Int`, `Bool`, `Int -> Bool`) with term level representations of types. We do this with a GADT:

```
data Type :: * -> * where
  TInt    :: Type Int
  TBool   :: Type Bool
  TArrow  :: Type a -> Type b -> Type (a -> b)
```

This will allow us to embed typing information into expressions that do not inherently carry information about their type. For example, it is impossible to determine the type of a variable by simply inspecting it, so we will annotate variables with type information. This information carries through to the type level so that GHC can validate that expressions are well typed for us!

Next, we need to define the type of boolean operators. We will also use a GADT for this since it will allow us to specify exactly what the input and output types should be for each operator

```
data Bop :: * -> * -> * where
  Add  :: Bop Int Int
  Sub  :: Bop Int Int
  Eq   :: Bop Int Bool
  Lt   :: Bop Int Bool
  Gt   :: Bop Int Bool
  And  :: Bop Bool Bool
  Or   :: Bop Bool Bool
```

Next, we are going to need a way to derive the term-level type of an STLC expression. In many cases, it will be possible to statically determine the type of an expression, however there will be no object to witness the type. To fix this, we will create a typeclass called `TypeOf`.

```
class TypeOf a where
  typeOf :: a -> Type a

instance TypeOf Int where
  typeOf _ = TInt

instance TypeOf Bool where
  typeOf _ = TBool
```

This type class gives us a way of obtaining a term-level representation of the type of an expression of type `Int` or `Bool`, the two base types in STLC. Notice how we don't actually care about the value of the expression; all we care about is the static type information known by GHC. GHC will use this type information to choose which implementation of `typeOf` to call, in turn giving us either `TInt` or `TBool`.

Now we are ready to define the actual expression data type.

```
data Expr :: * -> * where
  Lit     :: TypeOf a => a -> Expr a
  Var     :: String -> Type a -> Expr a
  Lambda  :: String -> Type a -> Expr b -> Expr (a -> b)
  App     :: Expr (a -> b) -> Expr a -> Expr b
  Bop     :: Bop a b -> Expr a -> Expr a -> Expr b
  If      :: Expr Bool -> Expr a -> Expr a -> Expr a
  Lift    :: a -> Type a -> Expr a
```

This definition is quite a bit more complicated than the original one we made. First, notice that there is only one constructor for literals. A literal can have any base type, denoted by the `TypeOf` class constraint. In this formulation, the expression `Lit 1` has type `Expr Int` and `Lit True` has type `Expr Bool`.

As mentioned above, we need to annotate variables with their types since there is no inherent way of determining the type of a variable out of context. Similarly, we also annotate the input to a lambda abstraction. The type of an STLC lambda expression is `Expr (a -> b)`, just like in Haskell!

Function application is fairly straightforward. In order to apply a function, it must have an arrow type, and the expression it is being applied to must have the correct input type. Binary operators are similar; two input expressions must be supplied that match the type of the operator input.

If statements work as you might expect. The first expression is the conditional, and it must have type `Bool`. The next two expressions are the “then” and “else” cases, and they must have the same type.

The final constructor is called `Lift`. This is a bit strange, and it didn't show up in our first formulation of STLC. `Lift` should be used when you have a value and you want to lift it into an expression, but it does not have a

`TypeOf` instance, and therefore you cannot construct a literal for it. This is not really useful when writing STLC programs, but it will be needed later in our interpreter.

Now let's look at some STLC programs! We will start with something really simple: adding two numbers. We can write the program `5 + 3` using the STLC abstract syntax as follows:

```
Bop Add (Lit 5) (Lit 3)
```

This expression has type `Expr Int`, just as we would expect! Now, what would happen if we try to add `5` and `True`?

```
Bop Add (Lit 5) (Lit True)
```

```
<interactive>:
    Couldn't match expected type ‘Int’ with actual type ‘Bool’
    In the first argument of ‘Lit’, namely ‘True’
    In the third argument of ‘Bop’, namely ‘(Lit True)’
```

We get a type error! If we had been doing this without GADTs, we would have had to check the types ourselves, but since the `Expr` data type is so expressive, GHC does the type checking for us! Cool!

Let's look at a few more simple programs.

```
plus :: Expr (Int -> Int -> Int)
plus = Lambda "x" TInt $ Lambda "y" TInt $ Bop Add (Var "x") (Var "y")

abs :: Expr (Int -> Int)
abs = Lambda "x" TInt $ If (Bop Lt (Var "x") (Lit 0))
               {- then -}  (Bop Sub (Lit 0) (Var "x"))
               {- else -}  (Var "x")
```

So, now we can write down well-typed STLC expressions. Great! But it would be nice if we could actually do something with them. Let's do the next sensible thing; write an interpreter.

## Getting Static Type Information

Before we write the full interpreter, let's devise a way to find the representation of the type of any expression. Since the type of any expression is known by GHC, this is definitely possible, but it will take a bit of work (but less work than typechecking!). Just like when we wanted to find the types of values, we will need a type class here.

```
class TypeOfExpr a where
    typeOfExpr :: Expr a -> Type a

instance TypeOfExpr Int where
    typeOfExpr _ = TInt

instance TypeOfExpr Bool where
    typeOfExpr _ = TBool
```

The instances of `TypeOfExpr` for base types are nearly identical to those of `TypeOf`. Again, we don't actually care about the values, only the staticly known type information. However, unlike base types, we need to deal with arrow types here. The instance for arrow types is a bit more complicated and relies on the structure of the expression.

```
instance TypeOfExpr b => TypeOfExpr (a -> b) where
    typeOfExpr (Var _ t)        = t
    typeOfExpr (Lambda _ t e)  = TArrow t $ typeOfExpr e
    typeOfExpr (App e1 _)      = case typeOfExpr e1 of
                                   TArrow _ t2 -> t2
```

```
typeOfExpr (If _ e1 _)     = typeOfExpr e1
typeOfExpr (Lift _ t)      = t
```

First, notice how the cases for `Lit` and `Bop` are omitted. We don't match over these because our formulation of STLC forbids literals and binary operations from having function types. The remaining cases are relatively straightforward. `Var`s and `Lift`s have type information stored in them, so it is trivial to retrieve their types. For `Lambda`s, we know the input type and we get the return type via a recursive call. The first expression in a function application is (by construction) an arrow type, therefore we can match on it to get the return type. If statements just have the type of one of the branches.

Notice that although we are inspecting types here, we are *not* type checking. If we were performing type checks then we would have to do things like make sure the argument to a function in the `App` case has the right type, make sure both branches of an `If` statement have the same type, etc.

## Proving Type Equality

The next thing that we like to add is the ability to *prove* the equality between two types. The `Data.Type.Equality` module defines the `(:~:)` type which can be thought of as a theorem (remember the Curry-Howard isomorphism?) stating the equality between two types. The only proof of this theorem is the data constructor `Refl`; meaning that any type is equal to itself.

Using `(:~:)`, let's write a function that returns a proof that two types are equal if such a proof exists.

```
eqTy :: Type u -> Type v -> Maybe (u :~: v)
eqTy TInt  TInt  = Just Refl
eqTy TBool TBool = Just Refl
eqTy (TArrow u1 u2) (TArrow v1 v2) = do
  Refl <- eqTy u1 v1
  Refl <- eqTy u2 v2
  return Refl
eqTy _     _     = Nothing
```

This states that `Int` is equal to `Int`, `Bool` is equal to `Bool`, and two arrow types are equal if their input and output types are equal.

## Substitution

The hardest part of writing an interpreter is the substitution case. Substitution occur as part of function application; when a lambda abstraction is applied to an argument, all occurences of the input variable in the function body get substituted for the argument. There are a couple of intricate cases in substitution, but the only one we really care about is `Var`. Let's take a look at the implementation of substitution.

```
subst :: String -> u -> Type u -> Expr t -> Expr t
subst _ _ _ (Lit b) = Lit b
subst x v u (Var y t)
    | x == y    = case eqTy u t of
                    Just Refl -> Lift v u
                    Nothing   -> error "ill-typed substitution"
    | otherwise = Var y t
subst x v u (Bop b e1 e2) = Bop b
                              (subst x v u e1)
                              (subst x v u e2)
subst x v u (If e1 e2 e3) = If (subst x v u e1)
                               (subst x v u e2)
                               (subst x v u e3)
subst x v u (Lambda y t e) | x == y    = Lambda y t e
                           | otherwise = Lambda y t (subst x v u e)
subst x v u (App e1 e2) = App (subst x v u e1) (subst x v u e2)
subst _ _ _ (Lift x t)  = Lift x t
```

First, notice the type signiture of `subst`. This function has four arguments. The first argument is the identifier of the variable we are substituting. The second argument is the value that we are substituting and the third is its type. The fourth argument is the expression we are substituting in. The function then returns resulting expression after the substitution.

One really interesting thing to note is that the expression that we are substituting in keeps the same type (`t`) before and after the substitution. One of the hardest parts of formally proving that a language is type safe is proving the Substitution Preserves Typing lemma. For our formulation of STLC, this property is built in to the type of the `subst` function. In this way, our implementation of `subst` acts as a proof of the Substitution Lemma.

Now let's turn our attention to the `Var` case. The first thing we do is check to see if this is indeed the variable that we were looking for. If it is, then we have to make sure that the variable has the right type to do the substitution. This is where we have to use the `eqTy` function. If we can get an equality proof then we can do the substitution safely.

We have a design decision to make in the case where there is no equality proof. We could decide that variable names are unique only to their type, and therefore we could multiple variables with the same name that are differentiated only by their types. In this case, we could return the original variable and not perform any substitution. The other design decision we could make is to throw an error since variables should be unique. This choice is more idiomatic, so this is what we do.

Let's now revisit the `eqTy` function. Was it necessary for this function to return a `Maybe (u :~: v)` or would a `Bool` be sufficient? After all, there are really only two values that a `Maybe (u :~: v)` could take on: `Just Refl` or `Nothing`. Could we just return `True` instead of `Just Refl` and `False` instead of `Nothing`?

We actually need the equality proof. Although a `Bool` may be enough to convince us that the substitution is legal, it is not enough to convince GHC. The equality proof carries additional information in its type that we cannot convey with a `Bool`.

## Interpreting STLC

We are finally ready to write our interpreter! The hardest part of the interpreting the language is substitution; the rest is pretty easy.

```
eval :: Expr t -> t
eval (Lit v)        = v
eval (Var _ _)      = error "Free variable has no value"
eval (Lambda x t e) = \v -> eval $ subst x v t e
eval (App e1 e2)    = (eval e1) (eval e2)
eval (Bop b e1 e2)  = (evalBop b `on` eval) e1 e2
eval (If b e1 e2)   | eval b    = eval e1
                    | otherwise = eval e2
eval (Lift x _)     = x

evalBop :: Bop a b -> a -> a -> b
evalBop Add = (+)
evalBop Sub = (-)
evalBop Eq  = (==)
evalBop Lt  = (<)
evalBop Gt  = (>)
evalBop And = (&&)
evalBop Or  = (||)
```

Notice that the type of the interpreter function is `Expr t -> t`. That means that the return type depends on the type of the input expression. For example, if the input is an `Int` expression, it will return an `Int`. If the input has an arrow type, the interpreter will actually return a Haskell function that we can evaluate in GHCi!

---

`Generated 2015-04-08 09:35:16.947024`