Type Wizardry

CIS 194 Week 13
25 November 2014

This week will be a sampling of the amazing things you can do with Haskell's type system. Unfortunately, mastering these skills takes time, effort, and lots of practice. If this course stretched forward for another semester, we would have the time to explore a lot of this in detail. However, we do not have that luxury of time. So, instead, consider this as a preview of what you can do with Haskell. The goal with the examples below is not to show you how to be a type expert; the goal is to show you what is possible with type expertise.

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

GADTs

A Generalized Algebraic Datatype (GADT) is a data structure that is *non-uniform* in its return type. GADTs use a different declaration syntax than do "regular" datatypes. Let's preview that syntax before looking at a real GADT:

```
data Maybe' a where
  Nothing' :: Maybe' a
  Just'    :: a -> Maybe' a

data List a where
  Nil  :: List a
  Cons :: a -> List a -> List a

data Bool' where
  True'  :: Bool'
  False' :: Bool'
```

The datatypes above are *not* GADTs, but they are written using GADT syntax. We label each constructor with its full type. Note that there is no more or less information in the declarations above than in the traditional datatype syntax; it's just different.

However, using GADT syntax, we can do something strange:

```
data G a where
  MkGInt  :: G Int
  MkGBool :: G Bool
```

Look at the return types: they're different! While that may not seem like much at first, check this out:

```
match :: G a -> a
match MkGInt  = 5
match MkGBool = False
```

`match` is a function that takes a `G a` as input and produces an `a` as output. As usual, this should work for *any* `a`. But, in the two equations above, we assume that the result is a number (`Int`, specifically) in the first equation and that the result is a `Bool` in the second. How is this possible?

The idea is that when we match on a GADT constructor, such as `MkGInt` or `MkGBool`, we learn something about the type `a`. Specifically, matching on `MkGInt` tells us that `a` *must* be `Int`. That's the whole point of putting `Int` in the return type of `MkGInt`! So, once we've matched on `MkGInt`, we now know that `a` is `Int`, and we can safely return `5` on the right-hand side of the equation. The case is similar in the second equation, where we learn that `a` is `Bool`.

It turns out that the GADT mechanism is very powerful. Below is a larger example.

First, we declare natural numbers (that is, integers greater than or equal to 0) using a unary notation, which turns out to be quite convenient:

```
data Nat = Zero | Succ Nat
```

A natural number is either 0 or the successor of some other natural number; for example, the nubmer 3 is encoded as `Succ (Succ (Succ Zero))`. Now, we define length-indexed vectors:

```
data Vec a n where
  VNil  :: Vec a Zero
  VCons :: a -> Vec a n -> Vec a (Succ n)
infixr 5 `VCons`     -- make `VCons` be *right*-associative
```

The first parameter, `a`, is the ordinary type parameter denoting the choice of element type – just like the parameter to `List`, above. The second parameter, `n`, denotes the length of the vector. Note that `VNil` requires its length to be `Zero`. `VCons`, on the other hand, says that its length is one more than the length of the tail of the vector.

We can build length-indexed vectors quite easily:

```
abc :: Vec Char (Succ (Succ (Succ Zero)))
abc = 'a' `VCons` 'b' `VCons` 'c' `VCons` VNil
```

Note that if we got the type wrong, the vector example wouldn't compile. This is the beauty of rich types: it lets us find more errors at compile time, instead of relying on runtime testing.

Type families

Once we have GADTs, it soon becomes necessary to do computation within types.

Consider the type of `vappend`, an append operation on vectors:

```
vappend :: Vec a n -> Vec a m -> Vec a ????????
```

We need to fill those question marks with the *sum* of `n` and `m` – a type-level addition operation.

First, let's look at a slightly simpler example:

```
type family Frob a where
  Frob Int  = Char
  Frob Bool = ()

quux :: G a -> Frob a
quux MkGInt  = 'x'
quux MkGBool = ()
```

A type family can be understood as a type function – a function from types to types. `Frob Int` is just `Char`, and `Frob Bool` is just `()`.

Note the return type of `quux`: it uses the `Frob` type family to compute th return type. In the first equation, GHC learns that `a` must be `Int`. GHC also knows that `Frob Int` is `Char`, so the `'x'` on the right-hand side is well typed. A similar analysis shows that the second equation is well typed.

Let's now return to length-indexed vectors. We'll need a type-level addition to proceed:

```
type family a + b where
  Zero   + b = b
  Succ a + b = Succ (a + b)
```

Note that the + we've just defined is totally independent from the normal + operator. This new one is on *types*.

Now, we can write `vappend`:

```
vappend :: Vec a n -> Vec a m -> Vec a (n + m)
vappend VNil         b = b
vappend (VCons h t) b = h `VCons` (vappend t b)
```

GHC has to do a lot of work to type-check that, but it works, by gum!

Larger examples

- We can use GADTs to refine the types of components of a large data structure. See a potential **re-design**

of Template Haskell and its **use**.

- We can use **propositional equality** (available as `Data.Type.Equality` in GHC 7.8) to be able to power a encoding of **dynamic typing** into Haskell.
- We can use the technique of **singletons** to gain (almost) all the power of dependent types (types that can depend on values). **Here** is a small example, including a "proof" of the associativity of addition.
- We can use rich types to enforce that modular numbers maintain a consistent modulus. **Here** is an example of modular numbers without compile-time guarantees. **This example** builds on top of the use of singletons, and **this one** uses GHC's built-in type-level naturals facility.
- Rich types allow us to encode even stronger invariants. The `units` package allows us to prevent code from adding, say, lengths to times. **Here** is a small example of the use of this package.
- A little type-level hackery allows us to define **functional extensionality for QuickCheck**, allowing QuickCheck users to define an equality check on whole functions.
- The power of rich types in Haskell goes quite far. **Here** is a somewhat inelegant encoding of a dependently-typed merge-sort in Haskell. The code is "proven" correct, in that a mistake in the implementation would prevent the code from compiling.

```
Generated 2014-12-04 13:37:39.902829
```