Type Wizardry

CIS 194 Week 10
1 April 2015

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 = O | S Nat
```

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

```
data Vec n a where
  Nil  :: Vec O a
  Cons :: a -> Vec n a -> Vec (S n) a
infixr 5 `Cons`     -- make `Cons` 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 (S (S (S O))) Char
abc = 'a' `Cons` 'b' `Cons` 'c' `Cons` Nil
```

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.

Recall the `head` function from the Haskell Prelude. This function is commonly viewed as a mistake sunce it is a *partial* function; it fails when the input list is empty. Instead, some people suggest using a safe version that returns the result as a `Maybe`. However, to use the value it is necessary to do a pattern match which is cumbersome and annoying. GADTs give us a way to encode the fact that a vector is non-empty in its type, and we can therefore write a safe head function that does not use `Maybe`s:

```
safeHead :: Vec (S n) a -> a
safeHead (Cons x _) = x
```

There are two key observations here. The first is that the length of the input vector is `S n`. In this expression, `n` could be any `Nat`, but since the `S` constructor is applied to it the length of the vector is at least 1. Becuase of this, we do not have to include a pattern match for the `Nil` case. In fact, including a match for the `Nil` case will give us a compiler error since the `Nil` constructor has type `Vec O a` which is not compatible with the input type `Vec (S n) a`. Now, if we try to call `safeHead` on an empty vector, we get a type error at compile time instead of a runtime error:

```
Couldn't match type ‘'O’ with ‘'S n0’
Expected type: List ('S n0) a
  Actual type: List 'O a
In the first argument of ‘head’, namely ‘Nil’
In the expression: head Nil
```

Now suppose we want to implement vector addition. The mathematical definition of vector addition states that the two vectors being added must have the same size. Using GADTs, we can encode this constraint in the type system.

```
(<+>) :: Num a => Vec n a -> Vec n a -> Vec n a
Nil <+> Nil = Nil
Cons x xs <+> Cons y ys = Cons (x + y) (xs <+> ys)
```

Notice how we do not need to match the cases where the first input is `Nil` and the second is a `Cons` and vice verse. The reason for this that we encode in the type of the function that both inputs have length `n`. This means that both vectors could be `Nil` or both could be `Cons`, but they can't be mixed.

We could take this one step further and define matrix multiplication where the type system requires that the inner dimensions of the matrices must match. This is left as an exercise.

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 n a -> Vec m a -> 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
  O   + b = b
  S a + b = S (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 n a -> Vec m a -> Vec (n + m) a
vappend Nil        b = b
vappend (Cons h t) b = h `Cons` vappend t b
```

GHC has to do a lot of work to type-check that, but it works, by gum! Now that we have an implementation of `vappend`, can we create an instance of the `Monoid` type class for vectors? Why or why not?

Theorem Proving

The Curry-Howard Isomorphism

There is a direct correspondence from theorems and proofs to types and programs. Programs act as proofs of their types and typecheckers verify these proofs. We have already seen this a little bit when we used GADTs to *prove* to the Haskell compiler that the input to `safeHead` was non-empty. Recall that the type of this function is: `Vec (S n) a -> a`. We could view this as a theorem stating that if there exists a non-empty vector of `a`s, then there exists an expression of type `a` and the implementation of `safeHead` is a proof of this theorem.

Propositional Logic

We can use the Haskell compiler to verify theorems in Propositional Logic. Propositional Logic is a field of mathematics that is concerned with proving *propositions*, or statements that could be either true or false. Propositions are related using *logical connectives*. Two of the most basic logical connectives are disjunction (or) and conjunction (and). We can encode these connectives in Haskell's type system. Consider the following definition of disjunctions:

```
-- Logical Disjunction
data p \/ q = Left  p
            | Right q
```

The disjunction has two constructors, `Left` and `Right`. The `Left` constructor takes in something of type `p` and the `Right` constuctor takes in something of type `q`. This means that a proposition of type `p \/ q` can be either a `p` or a `q` where `p` and `q` are themselves propositions. Now let's take a look at conjunctions:

```
-- Logical Conjunction
data p /\ q = Conj p q
```

Unlike disjunction, conjunction only has one constructor. This is because there is only one was to have a conjunction proposition; both `p` and `q` must be true.

Another basic logical connective is implication. Luckily, implication is already built in to the Haskell type system in the form of arrow types. The type `p -> q` is equivalent to the logical statement P implies Q. We can now prove a very simple theorem, Modus Ponens. Modus Ponens states that if P implies Q and P is true, then Q is true. We can write this down in Haskell:

```
modus_ponens :: (p -> q) -> p -> q
modus_ponens pq p = pq p
```

Notice that the type of `modus_ponens` is identical to the Prelude function `($)`. We can therefore simplify the proof:

```
modus_ponens' :: (p -> q) -> p -> q
modus_ponens' = ($)
```

This is a very interesting observation. The logical theorem Modus Ponens is exactly equivalent to function application!

In order for a logic to be consistent, there must be some proposition that in not provable. In our Haskell formulation of propositional logic, we will call this `False` and define it as follows.

```
data False
```

`False` is a datatype that has no constructors. This means that the type `False` in *uninhabited*; it is not provable since there is no way to write a program of type `False`. Now that we have a notion of `False`, we can define the logical connective not:

```
type Not p = p -> False
```

The `Not` type is really just an alias for `p -> False`. In other words, if `p` were true, then `False` would also be true. Now, let's prove another theorem, Modus Tollens. Modus Tollens states that if P imples Q and Q is not true, then P is not true.

```
modus_tollens :: (p -> q) -> Not q -> Not p
modus_tollens pq not_q = \p -> not_q $ pq p
```

This is a bit more involved than the proof of Modus Ponens above. Recall that `Not p` is really just a handy syntax for `p -> False`. This means that the proof of Modus Tollens should be a function that takes in an inhabitant of the proposition `p` and derives a contradiction from it. This is not too hard to do since we assume that `p` implies `q` and `q` implies `False`.

`Generated 2015-04-01 08:46:09.171432`

Powered by **shake**, **hakyll**, **pandoc**, **diagrams**, and **lhs2TeX**.