Parametric Polymorphism

CIS 194 Week 4 18 September 2014

While completing previous homework assignments, you probably spent a fair amount of time repeating yourself in certain parts of your code structure. For example, `wordsFrom` and `wordsFittingTemplate` in HW2 were awfully similar in structure. And the `MaybeLogMessage` and `MaybeInt` structures from HW3 are very similar. Can we abstract these patterns away and avoid this repetition in code? Sure we can!

Polymorphic data types

Let's take a look at (almost) two of the data structures from HW3:

```
data LogMessage = LogMessage Int String
data MaybeLogMessage = ValidLM LogMessage
                     | InvalidLM
data MaybeInt = ValidInt Int
              | InvalidInt
```

Those last two data structures are awfully similar. They both represent the possibility of failure. That is, they both optionally hold some type `a`; first, `a` is instantiated to `LogMessage`, and then to `Int`. It turns out that we can write this more directly:

```
data Maybe a = Just a
             | Nothing
```

(This definition isn't written using the > marks because it's part of the `Prelude`.)

A `Maybe a` is, possibly, an `a`. (`a` is called a *type variable* – it's a variable that stands in for a *type*.) So, instead of `MaybeLogMessage`, we could use `Maybe LogMessage`, and instead of `MaybeInt`, could use `Maybe Int`. `Maybe` is a *type constructor* or *parameterized type*. To become a proper, full-blooded type, we must supply `Maybe` with another type, like `LogMessage` or `Int`. When we do so, we simply replace all uses of `a` in `Maybe`'s definition with the type chosen as the parameter. Thus, the `Just` constructor of `Maybe Int` takes an `Int` parameter, and the `Just` constructor of `Maybe LogMessage` takes a `LogMessage` parameter.

Here is some sample code:

```
example_a :: Maybe Int -> Int
example_a (Just n) = n
example_a Nothing  = (-1)

example_b :: LogMessage -> Maybe String
example_b (LogMessage severity s) | severity >= 50 = Just s
example_b _                                        = Nothing
```

We're quite used to thinking about substituting terms in for other terms (that's what we use variables for!), and here we just apply this same principle to types.

Let's look at another example:

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

Given a type `t`, a `(List t)` consists of either the constructor `Empty`, or the constructor `Cons` along with a value of type `t` and another `(List t)`. Here are some examples:

```
lst1 :: List Int
lst1 = Cons 3 (Cons 5 (Cons 2 Empty))

lst2 :: List Char
lst2 = Cons 'x' (Cons 'y' (Cons 'z' Empty))

lst3 :: List Bool
lst3 = Cons True (Cons False Empty)
```

This `List` type is exactly like the built-in list type, only without special syntax. In fact, when you say `[Int]` in a type, that really means `[] Int` – allowing you to put the brackets around the `Int` is just a nice syntactic sugar.

Polymorphic functions

It's great that we can create polymorphic structures, but these become even more useful when we operate over them polymorphically. For example, let's say we want retrive the first element of a list. But, we don't know whether that list has any elements at all, and we need to be able to return *something* if we have an empty list. Our return type will thus be a `Maybe`. The type parameter should be left unspecified – that is, `a`:

```
safeHead :: List a -> Maybe a
safeHead Empty      = Nothing
safeHead (Cons x _) = Just x
```

Here, we have used `a` in a type signature in exactly the same way that we did when defining the `List` and `Maybe` types. It simply stands in for a type that will be specified later. To figure out what type should be used in place of `a` in a function, GHC performs *type inference* so you don't have to worry about getting it right.

Parametric polymorphism

One important thing to note about polymorphic functions is that **the caller gets to pick the types**. When you write a polymorphic function, it must work for every possible input type. This – together with the fact that Haskell has no way to directly make make decisions based on what type something is – has some interesting implications.

For starters, the following is very bogus:

```
bogus :: Maybe a -> Bool
bogus (Just 'x') = True
bogus _          = False
```

It's bogus because the definition of `bogus` assumes that the input is a `Maybe Char`. The function does not make sense for *any* value of the type variable `a`. On the other hand, the following is just fine:

```
isJust :: Maybe a -> Bool
isJust Nothing  = False
isJust (Just _) = True
```

The `isJust` function does not care what `a` is. It will always just make sense.

This "not caring" is what the "parametric" in parametric polymorphism means. All Haskell functions must be parametric in their type parameters; the functions must not care or make decisions based on the choices for these parameters. A function can't do one thing when `a` is `Int` and a different thing when `a` is `Bool`. Haskell simply provides no facility for writing such an operation. This property of a langauge is called *parametricity*.

There are many deep and profound consequences of parametricity. One consequence is something called *type erasure*. Because a running Haskell program can never make decisions based on type information, all the type information can be dropped during compilation. Despite how important types are when writing Haskell code, they are completely irrelevant when running Haskell code. This property gives Haskell a huge speed boost when compared to other languages, such as Python, that need to keep types around at runtime. (Type erasure is not the only thing that makes Haskell faster, but Haskell is sometimes clocked at 20x faster than Python.)

Another consequence of parametricity is that it restricts what polymorphic functions you can write. Look at this type signature:

```
strange :: a -> b
```

The `strange` function takes a value of some type `a` and produces a value of another type `b`. But, crucially, it isn't allowed to care what `a` and `b` are! Thus, *there is no way to write `strange`*!

```
strange = error "impossible!"
```

(The function `error`, defined in the `Prelude`, aborts your program with a message.)

What about

```
limited :: a -> a
```

That function must produce an `a` when given an `a`. There is only one `a` it can produce – the one it got! Thus, there is only one possible definition for `limited`:

```
limited x = x
```

In general, given the type of a function, it is possible to figure out various properties of the function just by thinking about parametricity. The function must have *some* way of producing the output type… but where could values of that type come from? By answering this question, you can learn a lot about a function.

Total and partial functions

Consider this polymorphic type:

```
[a] -> a
```

What functions could have such a type? The type says that given a list of things of type `a`, the function must produce some value of type `a`. For example, the Prelude function `head` has this type.

…But what happens if `head` is given an empty list as input? Let's look at the **source code** for `head`…

It crashes! There's nothing else it possibly could do, since it must work for *all* types. There's no way to make up an element of an arbitrary type out of thin air.

`head` is what is known as a *partial function*: there are certain inputs for which `head` will crash. Functions which have certain inputs that will make them recurse infinitely are also called partial. Functions which are well-defined on all possible inputs are known as *total functions*.

It is good Haskell practice to avoid partial functions as much as possible. Actually, avoiding partial functions is good practice in *any* programming language—but in most of them it's ridiculously annoying. Haskell tends to make it quite easy and sensible.

`head` is a mistake! It should not be in the `Prelude`. Other partial `Prelude` functions you should almost never use include `tail`, `init`, `last`, and `(!!)`. From this point on, using one of these functions on a homework assignment will lose style points!

What to do instead?

Replacing partial functions

Often partial functions like `head`, `tail`, and so on can be replaced by pattern-matching. Consider the following two definitions:

```
doStuff1 :: [Int] -> Int
doStuff1 []  = 0
```

```haskell
doStuff1 [_] = 0
doStuff1 xs  = head xs + (head (tail xs))

doStuff2 :: [Int] -> Int
doStuff2 []        = 0
doStuff2 [_]       = 0
doStuff2 (x1:x2:_) = x1 + x2
```

These functions compute exactly the same result, and they are both total. But only the second one is *obviously* total, and it is much easier to read anyway.

Writing partial functions

What if you find yourself *writing* a partial functions? There are two approaches to take. The first is to change the output type of the function to indicate the possible failure. Recall the definition of `Maybe`:

```haskell
data Maybe a = Nothing | Just a
```

Now, suppose we were writing `head`. We could rewrite it safely like `safeHead`, above. Indeed, there is exactly such a function defined in the **`safe` package**.

Why is this a good idea?

1. `safeHead` will never crash.
2. The type of `safeHead` makes it obvious that it may fail for some inputs.
3. The type system ensures that users of `safeHead` must appropriately check the return value of `safeHead` to see whether they got a value or `Nothing`.

OK, but what if we know that we will only use `head` in situations where we are *guaranteed* to have a non-empty list? In such a situation, it is really annoying to get back a `Maybe a`, since we have to expend effort dealing with a case which we “know” cannot actually happen.

The answer is that if some condition is really *guaranteed*, then the types ought to reflect the guarantee! Then the compiler can enforce your guarantees for you. For example:

```haskell
data NonEmptyList a = NEL a [a]

nelToList :: NonEmptyList a -> [a]
nelToList (NEL x xs) = x:xs

listToNel :: [a] -> Maybe (NonEmptyList a)
listToNel []     = Nothing
listToNel (x:xs) = Just $ NEL x xs

headNEL :: NonEmptyList a -> a
headNEL (NEL a _) = a

tailNEL :: NonEmptyList a -> [a]
tailNEL (NEL _ as) = as
```

You might think doing such things is only for chumps who are not coding super-geniuses like you. Of course, *you* would never make a mistake like passing an empty list to a function which expects only non-empty ones. Right? Well, there’s definitely a chump involved, but it’s not who you think.

Of course, some properties are more complex and are harder to encode in types. For example, it might be a critical property in your application that two lists are permutations of one another. What’s amazing is that Haskell’s type system is strong enough to encode such properties. Doing so is not for the faint of heart, and is beyond the scope of this course, but it’s tantalizing to know that such constructions are possible. If you want to learn more about this, come to Richard’s office hours, and he’ll be happy to tell you all about so-called *dependent types*, which can, in general, enforce arbitrary properties in types.

`Generated 2014-12-04 13:37:42.024283`