Polymorphism and Functional Programming Paradigms

CIS 194 Week 2
28 January, 2015

Additional Syntax

The introduction in the first class meeting did not discuss local variables. Here, we see a few examples of local variables in case these constructs are useful for you in writing your homework solutions.

`let` expressions

To define a local variable scoped over an expression, use `let`:

```
strLength :: String -> Int
strLength []     = 0
strLength (_:xs) = let len_rest = strLength xs in
                   len_rest + 1
```

In this case, the use of the local variable is a little silly (better style would just be `1 + strLength xs`), but it demonstrates the use of `let`. Don't forget the `in` after you've defined your variable!

`where` clauses

To define a local variable scoped over multiple guarded branches, use `where`:

```
frob :: String -> Char
frob []  = 'a'   -- len is NOT in scope here
frob str
  | len > 5   = 'x'
  | len < 3   = 'y'
  | otherwise = 'z'
 where
   len = strLength str
```

Note that the `len` variable can be used in any of the alternatives immediately above the `where` clause, but not in the separate top-level pattern `frob [] = 'a'`. In idiomatic Haskell code, `where` is somewhat more common than `let`, because using `where` allows the programmer to get right to the point in defining what a function does, instead of setting up lots of local variables first.

Haskell Layout

Haskell is a *whitespace-sensitive* language. This is in stark contrast to most other languages, where whitespace serves only to separate identifiers. (Haskell shares this trait with Python, which is also whitespace-sensitive.) Haskell uses indentation level to tell where certain regions of code end, and where new statements appear. The basic idea is that, when a so-called *layout herald* appears, GHC looks at the next thing it sees and remembers its indentation level. A later line that begins at the exact same indentation level is considered another member of the group, and a later line that begins at a lesser (more to the left) indentation level is not part of the group.

The layout heralds are `where`, `let`, `do`, `of`, and `case`. Because Haskell modules begin with `module Name where`, that means that the layout rule is in effect over the declarations in the file. This means that the following is no good:

```
x :: Int
x =
5
```

The problem is that the `5` is at the same indentation level (zero) as other top-level declarations, and so GHC considers it to be a new declaration instead of part of the definition of `x`.

The layout rule is often a source of confusion for newcomers to Haskell. But, if you get stuck, return to this decription (or, any of the many online) and re-read — often, if you think carefully enough about it, you'll see what's going on.

When calculating indentation level, tabs in code (you don't have any of these, do you?!?) are considered with tab stops 8 characters apart, regardless of what your editor might show you. This potential confusion is why tabs are a terrible, terrible idea in Haskell code.

Accumulators

Haskell's one way to repeat a computation is recursion. Recursion is a natural way to express the solutions to many problems. However, sometimes a problem's structure doesn't exactly match Haskell's structure. For example, say we have a list of numbers, that is, an `[Int]`. We wish to sum the elements in the list, but only until the sum is greater than 20. After that, the rest of the numbers should be ignored. Because recursion over a list builds up the result from the end backward, a naive recursion will not work for us. What we need is to keep track of the running sum as we go deeper into the list. This running sum is called an *accumulator*.

Here is the code that solves the stated problem:

```
sumTo20 :: [Int] -> Int
sumTo20 nums = go 0 nums   -- the acc. starts at 0
  where go :: Int -> [Int] -> Int
        go acc [] = acc   -- empty list: return the accumulated sum
        go acc (x:xs)
         | acc >= 20 = acc
         | otherwise = go (acc + x) xs


sumTo20 [4,9,10,2,8] == 23
```

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 :: [a] -> Bool
bogus ('X':_) = True
bogus _       = False
```

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

```
notEmpty :: [a] -> Bool
notEmpty (_:_) = True
notEmpty []    = False
```

The `notEmpty` 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.

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
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.

Recursion patterns

What sorts of things might we want to do with an `[a]`? Here are a few common possibilities:

- Perform some operation on every element of the list
- Keep only some elements of the list, and throw others away, based on a test
- "Summarize" the elements of the list somehow (find their sum, product, maximum…).
- You can probably think of others!

Map

Let's think about the first one ("perform some operation on every element of the list"). For example, we could add one to every element in a list:

```
addOneToAll :: [Int] -> [Int]
addOneToAll []     = []
addOneToAll (x:xs) = x + 1 : addOneToAll xs
```

Or we could ensure that every element in a list is nonnegative by taking the absolute value:

```
absAll :: [Int] -> [Int]
absAll []     = []
absAll (x:xs) = abs x : absAll xs
```

Or we could square every element:

```
squareAll :: [Int] -> [Int]
squareAll []     = []
squareAll (x:xs) = x^2 : squareAll xs
```

At this point, big flashing red lights and warning bells should be going off in your head. These three functions look way too similar. There ought to be some way to abstract out the commonality so we don't have to repeat ourselves!

There is indeed a way—can you figure it out? Which parts are the same in all three examples and which parts change?

The thing that changes, of course, is the operation we want to perform on each element of the list. We can specify this operation as a *function* of type `a -> a`. Here is where we begin to see how incredibly useful it is to be able to pass functions as inputs to other functions!

```
map :: (a -> b) -> [a] -> [b]
map _ []     = []
map f (x:xs) = f x : map f xs
```

We can now use `mapIntList` to implement `addOneToAll`, `absAll`, and `squareAll`:

```
exampleList = [-1, 2, 6]
```

```
map (+1) exampleList
map abs  exampleList
map (^2) exampleList
```

Filter

Another common pattern is when we want to keep only some elements of a list, and throw others away, based on a test. For example, we might want to keep only the positive numbers:

```
keepOnlyPositive :: [Int] -> [Int]
keepOnlyPositive [] = []
keepOnlyPositive (x:xs)
  | x > 0     = x : keepOnlyPositive xs
  | otherwise = keepOnlyPositive xs
```

Or only the even ones:

```
keepOnlyEven :: [Int] -> [Int]
keepOnlyEven [] = []
keepOnlyEven (x:xs)
  | even x    = x : keepOnlyEven xs
  | otherwise = keepOnlyEven xs
```

How can we generalize this pattern? What stays the same, and what do we need to abstract out?

The thing to abstract out is the *test* (or *predicate*) used to determine which values to keep. A predicate is a function of type `a -> Bool` which returns `True` for those elements which should be kept, and `False` for those which should be discarded. So we can write `filterIntList` as follows:

```
filter :: (a -> Bool) -> [a] -> [a]
filte _ [] = []
filter p (x:xs)
  | p x       = x : filter p xs
  | otherwise = filter p xs
```

Fold

We have one more recursion pattern on lists to talk about: folds. Here are a few functions on lists that follow a similar pattern: all of them somehow “combine” the elements of the list into a final answer.

```
sum' :: [Int] -> Int
sum' []     = 0
sum' (x:xs) = x + sum' xs

product' :: [Int] -> Int
product' [] = 1
product' (x:xs) = x * product' xs
```

```
length' :: [a] -> Int
length' []     = 0
length' (_:xs) = 1 + length' xs
```

What do these three functions have in common, and what is different? As usual, the idea will be to abstract out the parts that vary, aided by the ability to define higher-order functions.

```
fold :: (a -> b -> b) -> b  -> [a] -> b
fold f z []     = z
fold f z (x:xs) = f x (fold f z xs)
```

Notice how `fold` essentially replaces `[]` with `z` and `(:)` with `f`, that is,

```
fold f z [a,b,c] == a `f` (b `f` (c `f` z))
```

(If you think about `fold` from this perspective, you may be able to figure out how to generalize `fold` to data types other than lists…)

Now let's rewrite `sum'`, `product'`, and `length'` in terms of `fold`:

```
sum''     = fold (+) 0
product'' = fold (*) 1
length''  = fold addOne 0
 where addOne _ s = 1 + s
```

Of course, `fold` is already provided in the standard Prelude, under the name **`foldr`**. Here are some Prelude functions which are defined in terms of `foldr`:

- `length :: [a] -> Int`
- `sum :: Num a => [a] -> a`
- `product :: Num a => [a] -> a`
- `and :: [Bool] -> Bool`
- `or :: [Bool] -> Bool`
- `any :: (a -> Bool) -> [a] -> Bool`
- `all :: (a -> Bool) -> [a] -> Bool`

There is also **`foldl`**, which folds "from the left". That is,

```
foldr f z [a,b,c] == a `f` (b `f` (c `f` z))
foldl f z [a,b,c] == ((z `f` a) `f` b) `f` c
```

In general, however, you should use **`foldl'` from `Data.List`** instead, which does the same thing as `foldl` but is more efficient.

Functional Programming

We have seen now several cases of using a functional programming style. Here, we will look at several functions using a very functional style to help you get acclimated to this mode of programming.

Functional combinators

First, we need a few more combinators to get us going:

```
(.) :: (b -> c) -> (a -> b) -> a -> c
(.) f g x = f (g x)
```

The `(.)` operator, part of the Haskell Prelude, is just function composition. Say we want to take every element of a list and add 1 and then multiply by 4. Here is a good way to do it:

```
add1Mul4 :: [Int] -> [Int]
add1Mul4 x = map ((*4) . (+1)) x
```

While we're at it, we should also show the `($)` operator, which has a trivial-looking definition:

```
($) :: (a -> b) -> a -> b
f $ x = f x
```

Why have such a thing? Because `($)` is parsed as an operator, and this is useful for avoiding parentheses. For example, if we wish to negate the number of even numbers in a list, we could say

```
negateNumEvens1 :: [Int] -> Int
negateNumEvens1 x = negate (length (filter even x))
```

or

```
negateNumEvens2 :: [Int] -> Int
negateNumEvens2 x = negate $ length $ filter even x
```

No more parentheses!

Lambda

It is sometimes necessary to create an anonymous function, or *lambda expression*. This is best explained by example. Say we want to duplicate every string in a list:

```
duplicate1 :: [String] -> [String]
duplicate1 = map dup
  where dup x = x ++ x
```

It's a tiny bit silly to name `dup`. Instead, we can make an anonymous function:

```
duplicate2 :: [String] -> [String]
duplicate2 = map (\x -> x ++ x)
```

The backslash binds the variables after it in the expression that follows the `->`. For anything but the shortest examples, it's better to use a named helper function, though.

Currying and Partial Application

Remember how the types of multi-argument functions look weird, like they have "extra" arrows in them? For example, consider the function

```
f :: Int -> Int -> Int
f x y = 2*x + y
```

I promise that there is a beautiful, deep reason for this, and now it's finally time to reveal it: *all functions in Haskell take only one argument*. Say what?! But doesn't the function `f` shown above take two arguments? No, actually, it doesn't: it takes one argument (an `Int`) and *outputs a function* (of type `Int -> Int`); that function takes one argument and returns the final answer. In fact, we can equivalently write `f`'s type like this:

```
f' :: Int -> (Int -> Int)
f' x y = 2*x + y
```

In particular, note that function arrows *associate to the right*, that is, `W -> X -> Y -> Z` is equivalent to `W -> (X -> (Y -> Z))`. We can always add or remove parentheses around the rightmost top-level arrow in a type.

Function application, in turn, is *left*-associative. That is, `f 3 2` is really shorthand for `(f 3) 2`. This makes sense given what we said previously about `f` actually taking one argument and returning a function: we apply `f` to an argument `3`, which returns a function of type `Int -> Int`, namely, a function which takes an `Int` and adds 6 to it. We then apply that function to the argument `2` by writing `(f 3) 2`, which gives us an `Int`. Since function application associates to the left, however, we can abbreviate `(f 3) 2` as `f 3 2`, giving us a nice notation for `f` as a "multi-argument" function.

The "multi-argument" lambda abstraction

```
\x y z -> ...
```

is really just syntax sugar for

```
\x -> (\y -> (\z -> ...)).
```

Likewise, the function definition

```
f x y z = ...
```

is syntax sugar for

```
f = \x -> (\y -> (\z -> ...)).
```

This idea of representing multi-argument functions as one-argument functions returning functions is known as *currying*, named for the British mathematician and logician Haskell Curry. (His first name might sound familiar; yes, it's the same guy.) Curry lived from 1900-1982 and spent much of his life at Penn State—but he also helped work on ENIAC at UPenn. The idea of representing multi-argument functions as one-argument functions returning functions was actually first discovered by Moses Schönfinkel, so we probably ought to call it *schönfinkeling*. Curry himself attributed the idea to Schönfinkel, but others had already started calling it "currying" and it was too late.

If we want to actually represent a function of two arguments we can use a single argument which is a tuple. That is, the function

```
f'' :: (Int,Int) -> Int
f'' (x,y) = 2*x + y
```

can also be thought of as taking "two arguments", although in another sense it really only takes one argument which happens to be a pair. In order to convert between the two representations of a two-argument function, the standard library defines functions called `curry` and `uncurry`, defined like this (except with different names):

```
schönfinkel :: ((a,b) -> c) -> a -> b -> c
schönfinkel f x y = f (x,y)

unschönfinkel :: (a -> b -> c) -> (a,b) -> c
unschönfinkel f (x,y) = f x y
```

`uncurry` in particular can be useful when you have a pair and want to apply a function to it. For example:

```
Prelude> uncurry (+) (2,3)
5
```

Partial application

The fact that functions in Haskell are curried makes *partial application* particularly easy. The idea of partial application is that we can take a function of multiple arguments and apply it to just *some* of its arguments, and get out a function of the remaining arguments. But as we've just seen, in Haskell there *are no* functions of multiple arguments! Every function can be "partially applied" to its first (and only) argument, resulting in a function of the remaining arguments.

Note that Haskell doesn't make it easy to partially apply to an argument other than the first. The one exception is infix operators, which as we've seen, can be partially applied to either of their two arguments using an operator section. In practice this is not that big of a restriction. There is an art to deciding the order of arguments to a function to make partial applications of it as useful as possible: the arguments should be ordered from "least to greatest variation", that is, arguments which will often be the same should be listed first, and arguments which will often be different should come last.

Wholemeal programming

Let's put some of the things we've just learned together in an example that also shows the power of a "wholemeal" style of programming. Consider the function `foobar`, defined as follows:

```
foobar :: [Integer] -> Integer
foobar []      = 0
foobar (x:xs)
    | x > 3     = (7*x + 2) + foobar xs
    | otherwise = foobar xs
```

This seems straightforward enough, but it is not good Haskell style. The problem is that it is

- doing too much at once; and
- working at too low of a level.

Instead of thinking about what we want to do with each element, we can instead think about making incremental transformations to the entire input, using the existing recursion patterns that we know of. Here's a much more idiomatic implementation of `foobar`:

```
foobar' :: [Integer] -> Integer
foobar' = sum . map ((+2) . (*7)) . filter (>3)
```

This defines `foobar'` as a "pipeline" of three functions: first, we throw away all elements from the list which are not greater than three; next, we apply an arithmetic operation to every element of the remaining list; finally, we sum the results.

Notice that in the above example, `map` and `filter` have been partially applied. For example, the type of `filter` is

```
(a -> Bool) -> [a] -> [a]
```

Applying it to `(>3)` (which has type `Integer -> Bool`) results in a function of type `[Integer] -> [Integer]`, which is exactly the right sort of thing to compose with another function on `[Integer]`.

Point-free Style

The style of coding in which we define a function without reference to its arguments—in some sense saying what a function *is* rather than what it *does*—is known as "point-free" style. As we can see from the above example, it can be quite beautiful. Some people might even go so far as to say that you should always strive to use point-free style; but taken too far it can become extremely confusing. `lambdabot` in the `#haskell` IRC channel has a command `@pl` for turning functions into equivalent point-free expressions; here's an example:

```
@pl \f g x y -> f (x ++ g x) (g y)
join . ((flip . ((.) .)) .) . (. ap (++)) . (.)
```

This is clearly *not* an improvement!

Consider the following two functions:

```
mumble  = (`foldr` []) . ((:).)

grumble = zipWith ($) . repeat
```

Can you figure out what these functions do? What if I told you that they are both equivalent to the `map` function. These are great examples of how point-free style can be taken too far. For this reason, some people refer to it as point-less style.

```
Generated 2015-03-16 09:56:01.634659
```