Higher-order programming and type inference

CIS 194 Week 4
4 February 2013

Suggested reading:

- *Learn You a Haskell for Great Good* chapter "Higher-Order Functions" (Chapter 5 in the printed book; Chapter 6 online)

Anonymous functions

Suppose we want to write a function

```
greaterThan100 :: [Integer] -> [Integer]
```

which keeps only those `Integers` from the input list which are greater than 100. For example,

```
greaterThan100 [1,9,349,6,907,98,105] = [349,907,105].
```

By now, we know a nice way to do this:

```
gt100 :: Integer -> Bool
gt100 x = x > 100

greaterThan100 :: [Integer] -> [Integer]
greaterThan100 xs = filter gt100 xs
```

But it's annoying to give `gt100` a name, since we are probably never going to use it again. Instead, we can use an *anonymous function*, also known as a *lambda abstraction*:

```
greaterThan100_2 :: [Integer] -> [Integer]
greaterThan100_2 xs = filter (\x -> x > 100) xs
```

`\x -> x > 100` (the backslash is supposed to look kind of like a lambda with the short leg missing) is the function which takes a single argument `x` and outputs whether `x` is greater than 100.

Lambda abstractions can also have multiple arguments. For example:

```
Prelude> (\x y z -> [x,2*y,3*z]) 5 6 3
[5,12,9]
```

However, in the particular case of `greaterThan100`, there's an even better way to write it, without a lambda abstraction:

```
greaterThan100_3 :: [Integer] -> [Integer]
greaterThan100_3 xs = filter (>100) xs
```

`(>100)` is an *operator section*: if `?` is an operator, then `(?y)` is equivalent to the function `\x -> x ? y`, and `(y?)` is equivalent to `\x -> y ? x`. In other words, using an operator section allows us to *partially apply* an operator to one of its two arguments. What we get is a function of a single argument. Here are some examples:

```
Prelude> (>100) 102
True
Prelude> (100>) 102
False
Prelude> map (*6) [1..5]
[6,12,18,24,30]
```

Function composition

Before reading on, can you write down a function whose type is

```
(b -> c) -> (a -> b) -> (a -> c)
```

?

Let's try. It has to take two arguments, both of which are functions, and output a function.

```
foo f g = ...
```

In the place of the `...` we need to write a function of type `a -> c`. Well, we can create a function using a lambda abstraction:

```
foo f g = \x -> ...
```

`x` will have type `a`, and now in the `...` we need to write an expression of type `c`. Well, we have a function `g` which can turn an `a` into a `b`, and a function `f` which can turn a `b` into a `c`, so this ought to work:

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

(Quick quiz: why do we need the parentheses around `g x`?)

OK, so what was the point of that? Does `foo` actually do anything useful or was that just a silly exercise in working with types?

As it turns out, `foo` is really called `(.)`, and represents *function composition*. That is, if `f` and `g` are functions, then `f . g` is the function which does first `g` and then `f`.

Function composition can be quite useful in writing concise, elegant code. It fits well in a "wholemeal" style where we think about composing together successive high-level transformations of a data structure.

As an example, consider the following function:

```
myTest :: [Integer] -> Bool
myTest xs = even (length (greaterThan100 xs))
```

We can rewrite this as:

```
myTest' :: [Integer] -> Bool
myTest' = even . length . greaterThan100
```

This version makes much clearer what is really going on: `myTest'` is just a "pipeline" composed of three smaller functions. This example also demonstrates why function composition seems "backwards": it's because function application is backwards! Since we read from left to right, it would make sense to think of values as also flowing from left to right. But in that case we should write \((x)f \) to denote giving the value \(x\) as an input to the function \(f\). But no thanks to Alexis Claude Clairaut and Euler, we have been stuck with the backwards notation since 1734.

Let's take a closer look at the type of `(.)`. If we ask `ghci` for its type, we get

```
Prelude> :t (.)
(.) :: (b -> c) -> (a -> b) -> a -> c
```

Wait a minute. What's going on here? What happened to the parentheses around `(a -> c)`?

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 promised before 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 -> ...)).
```

Note, for example, that we can rewrite our composition function from above by moving the `\x -> ...` from the right-hand side of the `=` to the left-hand side:

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

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 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 (\x -> 7*x + 2) . 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]`.

This 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!

Folds

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

product' :: [Integer] -> Integer
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 :: b -> (a -> b -> b) -> [a] -> b
fold z f []     = z
fold z f (x:xs) = f x (fold z f 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 0 (\_ s -> 1 + s)
```

(Instead of `(\_ s -> 1 + s)` we could also write `(\_ -> (1+))` or even `(const (1+))`.)

Of course, `fold` is already provided in the standard Prelude, under the name `foldr`. The arguments to `foldr` are in a slightly different order but it's the exact same function. 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.

Higher-order programming and type inference

CIS 194 Week 4
4 February 2013

Suggested reading:

- *Learn You a Haskell for Great Good* chapter "Higher-Order Functions" (Chapter 5 in the printed book; Chapter 6 online)

Anonymous functions

Suppose we want to write a function

```
greaterThan100 :: [Integer] -> [Integer]
```

which keeps only those `Integers` from the input list which are greater than 100. For example,

```
greaterThan100 [1,9,349,6,907,98,105] = [349,907,105].
```

By now, we know a nice way to do this:

```
gt100 :: Integer -> Bool
gt100 x = x > 100

greaterThan100 :: [Integer] -> [Integer]
greaterThan100 xs = filter gt100 xs
```

But it's annoying to give `gt100` a name, since we are probably never going to use it again. Instead, we can use an *anonymous function*, also known as a *lambda abstraction*:

```
greaterThan100_2 :: [Integer] -> [Integer]
greaterThan100_2 xs = filter (\x -> x > 100) xs
```

`\x -> x > 100` (the backslash is supposed to look kind of like a lambda with the short leg missing) is the function which takes a single argument `x` and outputs whether `x` is greater than 100.

Lambda abstractions can also have multiple arguments. For example:

```
Prelude> (\x y z -> [x,2*y,3*z]) 5 6 3
[5,12,9]
```

However, in the particular case of `greaterThan100`, there's an even better way to write it, without a lambda abstraction:

```
greaterThan100_3 :: [Integer] -> [Integer]
greaterThan100_3 xs = filter (>100) xs
```

`(>100)` is an *operator section*: if `?` is an operator, then `(?y)` is equivalent to the function `\x -> x ? y`, and `(y?)` is equivalent to `\x -> y ? x`. In other words, using an operator section allows us to *partially apply* an operator to one of its two arguments. What we get is a function of a single argument. Here are some examples:

```
Prelude> (>100) 102
True
Prelude> (100>) 102
False
Prelude> map (*6) [1..5]
[6,12,18,24,30]
```

Function composition

Before reading on, can you write down a function whose type is

```
(b -> c) -> (a -> b) -> (a -> c)
```

?

Let's try. It has to take two arguments, both of which are functions, and output a function.

```
foo f g = ...
```

In the place of the `...` we need to write a function of type `a -> c`. Well, we can create a function using a lambda abstraction:

```
foo f g = \x -> ...
```

`x` will have type `a`, and now in the `...` we need to write an expression of type `c`. Well, we have a function `g` which can turn an `a` into a `b`, and a function `f` which can turn a `b` into a `c`, so this ought to work:

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

(Quick quiz: why do we need the parentheses around `g x`?)

OK, so what was the point of that? Does `foo` actually do anything useful or was that just a silly exercise in working with types?

As it turns out, `foo` is really called `(.)`, and represents *function composition*. That is, if `f` and `g` are functions, then `f . g` is the function which does first `g` and then `f`.

Function composition can be quite useful in writing concise, elegant code. It fits well in a "wholemeal" style where we think about composing together successive high-level transformations of a data structure.

As an example, consider the following function:

```
myTest :: [Integer] -> Bool
myTest xs = even (length (greaterThan100 xs))
```

We can rewrite this as:

```
myTest' :: [Integer] -> Bool
myTest' = even . length . greaterThan100
```

This version makes much clearer what is really going on: `myTest'` is just a "pipeline" composed of three smaller functions. This example also demonstrates why function composition seems "backwards": it's because function application is backwards! Since we read from left to right, it would make sense to think of values as also flowing from left to right. But in that case we should write \((x)f \) to denote giving the value \(x\) as an input to the function \(f\). But no thanks to Alexis Claude Clairaut and Euler, we have been stuck with the backwards notation since 1734.

Let's take a closer look at the type of `(.)`. If we ask `ghci` for its type, we get

```
Prelude> :t (.)
(.) :: (b -> c) -> (a -> b) -> a -> c
```

Wait a minute. What's going on here? What happened to the parentheses around `(a -> c)`?

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 promised before 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 -> ...)).
```

Note, for example, that we can rewrite our composition function from above by moving the `\x -> ...` from the right-hand side of the `=` to the left-hand side:

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

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 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 (\x -> 7*x + 2) . 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]`.

This 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!

Folds

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

product' :: [Integer] -> Integer
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 :: b -> (a -> b -> b) -> [a] -> b
fold z f []     = z
fold z f (x:xs) = f x (fold z f 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 0 (\_ s -> 1 + s)
```

(Instead of `(\_ s -> 1 + s)` we could also write `(\_ -> (1+))` or even `(const (1+))`.)

Of course, `fold` is already provided in the standard Prelude, under the name `foldr`. The arguments to `foldr` are in a slightly different order but it's the exact same function. 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.