Lesson 9

HIGHER-ORDER FUNCTIONS

June 13, 2023

1. Higher-Order Functions
2. The HOF Zoo
3. Mathematics of Higher-Order Functions

Last time, we learned about the concept of **parametric polymorphism**.

We saw how polymorphic types could be parameterized by **type variables**, which allowed values of polymorphic type to be used in generic ways, by instantiating them at different types at each use site.

We also saw how we could define our own polymorphic datatypes, and rederived the true definition of `option` and `list` from it.

We then experimented with polymorphic sorting, where we could create a sorting function on lists which operated *generically* in the type of the list's elements, by providing a comparison function as another input to the sorting function.

1 - Higher-Order Functions

We saw how, with polymorphic types and polymorphic functions, we have a certain concept of parameterization. A polymorphic type denoted not just a single type, but *many* different types. Similarly, a polymorphic function denoted not a single function, but a template for many functions of different types.

Each of these functions essentially did the same thing, however[1]. In this lecture, we'll explore **higher-order functions**, which allows us to capture common design patterns in code to create families of functions which all do different, but related, things.

[1] They had to, in order to be polymorphically generalizable!

In the last lecture, we proposed a polymorphic sorting function that took in a comparison function. It looked like:

```
fun sort (cmp : 'a * 'a -> order, L : 'a list) = (* ... *)
```

This `sort` function doesn't do the "same essential thing", however, its behavior depends on the `cmp` function it receives as input!

It turns out that, while the `sort` function is polymorphic, the key fact is that it is *also* a higher-order function!

When writing functions so far this semester, we've mostly looked at functions which take in tuple values, and return values of base type, or other slightly more interesting types like `option`s, `list`s, or `tree`s.

These functions are **first-order**. They are functions in the classic, intuitive sense.

But it is possible for a function to return a function – or for that matter, take in a function, as well.

Def A **higher-order function** is a function which takes in functions or returns functions.

`sort` is thus a higher-order function, due to taking in the comparison function `cmp`.

We could write code like so:

```
fun mod12Compare (x, y) = Int.compare (x mod 12, y mod 12)

val sorted = sort (mod12Compare, [4, 3, 1, 2])
```

This is kind of verbose, though. Do we need to do a `fun` declaration every time that we want to specify our comparison function?

Luckily, the answer is no. Recall lambda expressions, which are anonymous function values. We haven't used them extensively, but it turns out something they are very useful for is making quick functions as arguments to other functions.

Suppose we wanted to sort a list, modulo 12. Then we might write:

```
val L = [4, 3, 1, 2]
val sorted =
  sort (fn (x, y) => Int.compare (x mod 12,  y mod 12), L)
```

We can also return functions, not just take them in.

Def We call a function which returns another function **curried**.
Suppose we have our `add` function.

```
(* add : int * int -> int *)
fun add (x, y) = x + y
```

What if instead of taking in a tuple of both `int`s to be added, we returned a lambda expression which took in the second?

```
(* cadd : int -> int -> int *)
fun cadd x = fn y => x + y
```

Note Type arrows are *right-associative*. This means the type `int -> int -> int` means the same as `int -> (int -> int)`.

It’s a little bit of a pain to have to explicitly write out the `fn y => x + y` that we return, however.

SML helps, by having some syntactic sugar for writing curried functions.

The following two definitions are equivalent:

```
(* cadd : int -> int -> int *)
fun cadd x = fn y => x + y
fun cadd x y = x + y
```

In general, you can always add more arguments (separated by spaces), which SML will understand to mean "return a function which takes in that argument as a parameter". This generalizes to many arguments, too.

Here, we say that `cadd` is a *curried* form of `add`! It differs from `add` in usage in that arguments are passed in one by one.

```
val 2 = add (1, 1)
val 2 = cadd 2 2
```

Note that function application is *left-associative*, meaning that `add 2 2` is the same as `(add 2) 2`.

Seems these functions do the same thing. Are they extensionally equivalent?

The answer is no! `add` and `cadd` don't even have the same type. **Two expressions can only be extensionally equivalent if they have the same type**.

So `add` and `cadd` aren't extensionally equivalent, but they do "essentially the same thing".

The main advantage of `cadd` in this scenario is that it can take in its arguments separately. We will see later why this is a virtue, but first we have some more HOFs to learn about.

2 - The HOF Zoo

Abstraction

Abstraction is the name of the game in computer science.

We abstracted away bits and bytes so that we could think about data and programs. We abstracted away unrestricted control flow for structured constructs so that we could better reason about those programs, and we added specifications and types so that we could better communicate what our programs do.

With higher-order functions, we can abstract code over code itself. We've seen this once, by writing a `sort` function which varies depending on the code of the corresponding `cmp` function.

This prevents us from having to rewrite multiple `sort` functions with the same "core logic". Let's see some examples of other HOFs which reduce common logic.

Sometimes we're interested in applying some transformation to every element of a list.

```
fun incrementAll [] = []
  | incrementAll (x::xs) = (x + 1) :: incrementAll xs

fun toStringAll [] = []
  | toStringAll (x::xs) = Int.toString x :: toStringAll xs
```

If we take away the operation that we perform to each element `x`, the underlying function looks exactly the same!

We will capture this phenomenon with a HOF called `map`.

```
map : ('a -> 'b) -> 'a list -> 'b list
```
REQUIRES: `true`
ENSURES: `map f [x1, ..., xn]` $\cong$ `[f x1, ..., f xn]`

```
fun map (f : 'a -> 'b) ([] : 'a list) : 'b list = []
  | map f (x::xs) = f x :: map f xs
```

Then we obtain that `incrementAll` $\cong$ `map (fn x => x + 1)`, and `toStringAll` $\cong$ `map Int.toString`.

The main strength of currying is in **partial application**.

Def **Partial application** is the act of applying *some* of the curried arguments to a curried function, but not all.

Partial application lets us obtain increasingly-specific instances of a higher-order function, which acts as a *template* for a family of functions that all behave the same.

In this case, `map` is the general design for a family of functions that entail transforming elements of a list, and `incrementAll` and `toStringAll` are concrete instances of this design! So instead, we could write:

```
val incrementAll = map (fn x => x + 1)
val toStringAll = map Int.toString
```

Why does writing the above work? Consider the difference between

```
fun incrementAll L = map (fn x => x + 1) L
val incrementAll = map (fn x => x + 1)
```

Remember, we can do the latter because functions are values. Both ways, we end up with `incrementAll : int list -> int list`.

The first declaration is a function which explicitly names its argument, `L`, and then when given `L`, evaluates to `map (fn x => x + 1) L`.

The second declaration, however, is a value which is a function, and can be put next to any argument, such as `L`. If we were to write `incrementAll L`, then by the definition of `incrementAll`, we would have `map (fn x => x + 1) L`, which is the same thing!

This is a general law called **eta expansion**.

Def We say that for any function `f : t1 -> t2`, the lambda expression `fn x => f x` is the **eta-expanded** version of `f`.

The key observation is that `f` and `fn x => f x` are both extensionally equivalent. They mean the same thing.

Another way of looking at it is that functions don't *need* to name their arguments – functions already expect their arguments.

It's now that we can concretize our claim that `sort`, as defined previously, defines a "family of functions".

We can do so by defining a new curried sort, like so:

```
fun sortCurried cmp L = sort (cmp, L)
```

They look similar, but the advantage is that now, we can see that the concrete instances of `sort`, given its comparison function, define every possible sorting function on lists!

```
val intSort = sortCurried Int.compare
val stringSort = sortCurried String.compare
val mod12Sort =
  sortCurried (fn (x, y) => x mod 12 < y mod 12)
```

The power of higher-order functions is in being able to define functions which generalize entire code patterns, that essentially automate the process of coding for you.

There is almost a spiritual component to the definition of these `map` and `sort` functions, in that they inherently carry the Platonic structure of transforming a list of data, and sorting a list of data respectively.

In the large, programming becomes the recognition and manipulation of these archetypes, specifying them to fit your given use case. Seen in this way, higher-order functions pave the way to, indeed, *every function ever*.

Another common pattern is keeping only the elements of a list that satisfy some predicate. This predicate might vary depending on the use case, but the overall pattern remains the same.

This leads us to a HOF named `filter`.

`filter : ('a -> bool) -> 'a list -> 'a list`
REQUIRES: `true`
ENSURES: `filter p xs` evaluates to all elements `x` in `xs` such that `p x` $\cong$ `true`, in the same order

Filter can be implemented like so:

```
fun filter (p : 'a -> bool) ([] : 'a list) : 'a list = []
  | filter p (x::xs) =
      if p x then
        x :: filter p xs
      else
        filter p xs
```

So, for instance, we have that:

```
val isEven = fn x => x mod 2 = 0
val keepEvens = filter isEven
val [2, 4] = keepEvens [1, 2, 3, 4]
val keepOdds = filter (fn x => not (isEven x))
val [1, 3] = keepOdds [1, 2, 3, 4]
```

It's a little ugly to have to write `fn x => not (isEven x)`, though.

Some functional programmers would rather be hit by a bus than have to write out an explicit lambda expression, if it can be avoided.

Though we can choose to write something like `map (fn x => f x) xs`, as opposed to `map f xs`, generally it is agreed upon that the former is ugly, and the latter is more "clean".

We will now see something which will help us achieve this style of programming.

Something you learn about early on in mathematics is *function composition*.

In SML, functions are meant to be closer to their mathematical counterparts. We can define a notion of function composition for SML functions too!

We want a function which takes in two functions and essentially strings them together. We don't know what their types are, so we will simply call the first one `'a -> 'b`, and the second `'b -> 'c`. Whatever the first one returns, the second one needs to take as input.

Our output will be a function which takes in an input, passes it into the first function, and then passes the result of that into the second function.

Given `'a -> 'b` and `'b -> 'c`, that function's type must be `'a -> 'c`.

`compose : ('b -> 'c) * ('a -> 'b) -> 'a -> 'c`
REQUIRES: `true`
ENSURES: `compose (g, f)` is such that `compose (g, f) x` $\cong$ `g (f x)`, for all `x`

Let's write it!

```
fun compose (g : 'b -> 'c, f : 'a -> 'b) : 'a -> 'c =
  fn x => g (f x)
```

We take the function arguments in reverse, to be more in line with the mathematical notation, writing $g \circ f$ to be the composition of g with f, such that f is applied first.

In SML, we also have `o` defined as the infix composition operator. So instead of `compose (g, f)`, we could more tersely write `g o f`.

So now, instead of writing `fn x => not (isEven x)`, we can write:

```
val isOdd = not o isEven
```

because it is extensionally equivalent.

Much more terse!

There is one final pattern that is common to working with lists.

Oftentimes, we are interested in "summarizing" the data in a list. We are interested in iterating over the elements of a list, producing some value which changes for every element that we see.

This takes the form of, for instance, summing all the elements of an `int list`, or concatenating all the strings in a `string list`.

For instance:

```
fun sum [] = 0
  | sum (x::xs) = x + sum xs

fun concat [] = ""
  | concat (x::xs) = x ^ concat xs

fun flatten [] = []
  | flatten (x::xs) = x @ flatten xs
```

These all look kind of similar!

All of these functions have a common root, in that they have some "initial value" that is returned upon the empty list, and which is otherwise transformed by some common operation, in conjunction with each element of the list.

In a sense, it looks very similar to this common pattern in other programming languages:

```
acc = default

for x in xs:
  acc = f(x)
```

Let's write it in SML!

We call this process **folding**. We will implement a function, `foldl`, which involves traversing the list from left to right, and transforming an accumulator value.

The type of this function will be `('a * 'b -> 'b) -> 'b -> 'a list -> 'b`.

These can be broken down into four parts:

- `'a * 'b -> 'b` - the "transforming function", which acts upon the accumulator and each fresh value of the list
- `'b` - the "default value" which serves as the initial accumulator
- `'a list` - the list to be "folded"
- `'b` - the final value to be returned, of the same type as the accumulator

```
foldl : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
```

REQUIRES: `true`

ENSURES: `foldl f z [x1, ..., xn]` evaluates to `f (xn, ... f (x2, f (x1, z)) ...)`

```
fun foldl f z [] = z
  | foldl f z (x::xs) = foldl f (f (x, z)) xs
```

Essentially, when we run out of elements in the list, we simply return what we have accumulated so far.

Otherwise, we update our accumulator and keep recursing through the list.

A Left Folding Example

How can we recover some of the common list functions we wrote earlier, using `foldl`?

Let's implement a function which sums over all of the elements of an `int list`:

```
fun sum L = foldl (op+) 0 L
```

How does it work?

Let's try using our `sum` function.

```
= sum [1, 2, 3]
= foldl (op+) 0 [1, 2, 3]
= foldl (op+) (1 + 0) [2, 3]
= foldl (op+) 1 [2, 3]
= foldl (op+) (1 + 2) [3]
= foldl (op+) 3 [3]
= foldl (op+) (3 + 3) []
= foldl (op+) 6 []
= 6
```

That's not the only way to fold a list, however. What if we want to fold a list from right to left? Let's implement `foldr`, of the same type.

```
foldr : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
```
REQUIRES: `true`
ENSURES: `foldr f z [x1, ..., xn]` evaluates to
`f (x1, f (x2, ... f (xn, z) ...))`

```
fun foldr f z [] = z
  | foldr f z (x::xs) = f (x, foldr f z xs)
```

The main way to remember how to implement the two folds is in when we combine with the first element, `x`.

```
fun foldl f z [] = z
  | foldl f z (x::xs) = foldl f (f (x, z)) xs

fun foldr f z [] = z
  | foldr f z (x::xs) = f (x, foldr f z xs)
```

In `foldl`, due to eager evaluation, the first thing that happens is we apply `f` to `x`! This corresponds to going *left-to-right*, as we want to transform the first element first.

In `foldr`, due to eager evaluation, we do the application of `f` to `x` *last*. For similar reasons, this corresponds to going right-to-left.

So now, how do we use `foldl` and `foldr` to implement `sum`, `concat`, and `flatten`?

The simple way is simply to visualize the accumulator changing, by applying the function to the elements and accumulator, going left to right or right to left. Recall that the transform function `f` always takes the accumulator as its second argument.

```
val sum = foldl (op+) 0
val concat = foldr (op^) ""
val flatten = foldr (op@) []
```

It is said that `foldr` is the "natural" fold[2]. We end up with all the elements in order, merely joined by the transformation function.

Let's try `foldr (op^) "" ["I", "LOVE", "150"]`.

```
= foldr (op^) "" ["I", "LOVE", "150"]
= "I" ^ foldr (op^) "" ["LOVE", "150"]
= "I" ^ ("LOVE" ^ foldr (op^) "" ["150"])
= "I" ^ ("LOVE" ^ ("150" ^ foldr (op^) "" []))
= "I" ^ ("LOVE" ^ ("150" ^ ""))
= "ILOVE150"
```

[2] For more, consult Frank Pfenning's excellent document `http://www.cs.cmu.edu/~me/courses/15-150-Spring2020/lectures/10/origami.pdf`

For instance, let's try `foldl (op::) [] [1, 2, 3]`. What do you expect to happen?

```
= foldl (op::) [] [1, 2, 3]
= foldl (op::) [1] [2, 3]
= foldl (op::) [2, 1] [3]
= foldl (op::) [3, 2, 1] []
= [3, 2, 1]
```

We see that we end up with, essentially, `3 :: 2 :: 1 :: []`. This applied the transformation function to each element, but in reverse!

If we look at the specification of `foldl`, though, this is exactly what it purported to do. We expected to see `f (xn, ... f (x2, f (x1, z)) ... )`.

So `rev` can be reimplemented as `foldl (op::) []`!

3 - Mathematics of Higher-Order Functions

Equivalence

λ

At this point, we might be wondering about the implications for mathematical analysis of functions when higher-order functions get involved. In particular, how does equivalence get affected?

Recall our definition for extensional equivalence on functions, for `f : t1 -> t2` and `g : t1 -> t2`. We require that, for all values `x : t1`, that `f x` $\cong$ `g x`.

It turns out, no extra machinery is necessary! We already have a definition for when functions should be $\cong$ (which is exactly the above), meaning that it's no problem if `t2` is a function type.

It might seem a little different, however, when dealing with when `t1` is a function type. How do we reason about if two function values are the "same"?

Is `fn (x, y) => x + y` the same as `fn (y, x) => y + x`?

What about `fn x => x + x` versus `fn x => 2 * x`?

Fortunately, it doesn't matter. While we specified that, for all values, `f` and `g` behave the same, because of referential transparency, we identify values by whether or not they are extensionally equivalent. So this is equivalent to saying:

If, for $\texttt{x} \cong \texttt{y}$, then $\texttt{f x} \cong \texttt{g y}$. This goes both ways, so we get that two extensionally equivalent HOFs behave the same on extensionally equivalent arguments.

So we know that, for instance, as a consequence of the fact that `fn x => 2 * x` and `fn x => x + x` are equivalent, then `map (fn x => 2 * x)` and `map (fn x => x + x)` must be equivalent as well.

Even though HOFs generalize over other arbitrary code, we have a mathematical guarantee that "equals-for-equals" still holds! HOFs then, in a sense, still act generically over their inputs, in a way that respects extensional equivalence.

A free consequence that comes out of this: easy refactoring. Functions being used in a higher-order context can be updated without fear of breaking a higher-order codebase, so long as each function individually remains extensionally equivalent.[3]

[3] This is *really useful*.

In this lecture, we saw how we could write **higher-order functions**, which were functions which generalized over other functions, possibly being able to be specialized to arbitrary precision by splitting up arguments into curried form.

With functions like `map` and `foldl`, we can think of them as defining a hierarchy of functions, all of which are defined from a common ancestors. Thus, the descendants of `map` might be `incrementAll` and `toStringAll`, and the descendants of `foldl` might be `sum`, `concat`, and `flatten`.

With this hereditary understanding of functions, we can abstract away even design patterns in code, reducing boilerplate and overall achieving a more holistic understanding.

Thank you!