Lesson 10
COMBINATORS AND STAGING

June 16, 2023

Lesson Plan

λ

Last time

Last time, we went over **higher-order functions**, which are functions which can take in functions as input and return other functions.

We learned that **curried** functions take in multiple arguments at separate times, by taking them in one-by-one, and returning functions which take in the rest.

We also learned about the menagerie of HOFs that we will use in this class, being `map`, `filter`, `o`, `foldl`, `foldr`.

1 - Staging

Currying

What is the advantage of currying?

Before, we talked about how we can use currying to define hierarchies of functions.

We can instantiate functions we'd ordinarily have to write recursive function definitions for, by using HOFs like `map` as templates, and producing expressions such as `map Int.toString`.

Currying is then important because each curried argument represents a *specialization* of a function!

- `foldr` - general template for accumulating in a list
- `foldr (op+)` - general template for summing elements in a list, plus a value
- `foldr (op+) 0` - function for finding sum of a list

Trivial Totality

A `fun fact` is that syntactically sugared curried functions are trivially total.[1]

Consider the definition of a function like `map`:

```
fun map f [] = []
  | map f (x::xs) = f x :: map f xs
```

Is `map` total? I claim – yes!

But now, we also know that `map f [1, 2, 3]` should be extensionally equivalent to `[f 1, f 2, f 3]`, and `f` might not be a total function. What gives?

Indeed, if we evaluate `map (fn x => x div 0) [1, 2, 3]`, we get a raised exception `Div`. What gives?

[1] An even more fun fact is that there were three alliterations in that sentence.

Trivial Totality

Key Fact We said that `map` was total – this does not necessarily say anything about the totality of `map f`! These are *two different functions*.

Consider the definition of `map`, which can be desugared to the following:

```
val map = fn f => fn [] => (* ... *) | x::xs => (* ... *)
```

It is easy to see that for any function value `f : t1 -> t2`, `map f` immediately evaluates to a lambda expression, which takes in a list and evaluates to either case of `map`.

This is an important conceptual distinction to keep in your mind. So we say that a curried function like `map` is "trivially total".

Trivial Totality?

A question remains then - is every value `t1 -> t2 -> t3` total?

The answer: *no!* Just because `map` and friends are, doesn't mean all such values of curried type are. For instance, take the following example:

```
fun loop () = loop ()

fun f x =
  let
    val x = loop ()
  in
    fn y => 0
  end
```

This function does not immediately return a lambda expression upon being given a value. It actually immediately evaluates `loop ()`, an infinite loop.

A Carnegie Mellon Example

Suppose that you are building a booth.[2]

It's the day of move-on, and you still haven't finished painting the wall boards. Your good friend stays behind to get them painted, and you refuse to move the rest of the booth to Midway until they're done.

That is silly.

[2]For readers unfamiliar with Carnegie Mellon traditions, this is a yearly festival where undergraduate students become construction workers and build small houses as decorations for alumni. I'm not joking.

The Logistics of Booth Building

When building a booth, painting the walls is necessary, but comes much after other steps, like setting up the floorboards, constructing the walls, and moving the wood to Midway in the first place!

The point: It’s absurd to wait on something completely unrelated, when you could do the work now with what you have!

Staging

We refer to this as **staging**.

Def We use the term **staging** to describe the act of deliberately placing computations at certain points with respect to receiving curried arguments .

So instead of saving all computations for when all the curried arguments are received, we can instead move some computations to when only the *necessary* arguments have been received.

Suppose we have the following function:

```
val f = fn x => fn y => x + y
```

Can we move the expression `x + y` any earlier in the curried function? The answer is *no*, because `x + y` depends on both arguments!

A Code Example

This comes up all the time!

Consider the following artificial code example:

```
fun mystery x y =
  let
    val res = horrible_computation x
  in
    res + y
  end
```

`horrible_computation` takes 3 years to evaluate.

A Code Example

```
fun mystery x y =
  let
    val res = horrible_computation x
  in
    res + y
  end
```

3 years is kind of a long time. Suppose we're interested in evaluating the following:

```
val res1 = mystery 2 4
val res2 = mystery 1 2
val res3 = mystery 2 5
```

This code takes 9 years in total, to run. But it doesn't need to!

Speed Gains

Something we notice about `mystery` is that `horrible_computation` doesn't actually depend on `y`.

So we can rewrite it as:

```
fun mystery2 x =
  let
    val res = horrible_computation x
  in
    fn y => res + y
  end
```

Now instead of returning a lambda which accepts `y`, computes the horrible computation, and then returns, we first compute the horrible computation!

Speed Gains

This lets us write:

```
val f = mystery2 2
val g = mystery2 1
val res1 = f 4
val res2 = g 2
val res3 = f 5
```

We can't avoid the cost of computing `mystery2` twice, but 6 years isn't so long.

Benefits of Staging

Usually, the order in which computations happen doesn’t matter, due to extensional equivalence. We might care about things which go beyond simple extensional equivalence however, such as expensive computations, or things that break extensional equivalence, such as side effects or mutability.

Later this semester, we will see how side effects can make correct knowledge of staging even more essential.

2 - Cost Analysis of HOFs

Cost Analysis of HOFs

We've written some higher-order functions at this point, and we're fairly convinced of their correctness, but what can we say about their efficiency?

Take `map` for example.

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

From first glance, it looks like the recurrence is a standard recursion on a list, and thus comes out to a bound of $O(n)$. When analyzing the recursive case, a question comes to mind, however – what is the work of `f`?

The Cost of Code

Because HOFs like `map` are code which is parameterized on other code, the run-time cost of functions like `map` is also parameterized by the cost of the input function!

This is different than before HOFs, when we only had to deal with being passed values that didn't have any notion of cost associated with them – they just were.

In this case, we would say that the cost of `map` is $O(n)$ in the number of calls to `f`, but we can't really say anything better than that.

3 - HOFs and Trees

Mapping and Folding

We've so far seen mapping and folding on lists. These aren't notions that are specific to lists, however.[3]

Suppose we have a polymorphic tree type, as we've defined before, such as:

```
datatype 'a tree = Empty | Node of 'a tree * 'a * 'a tree
```

We are interested in a `map` function which transforms every element of the tree, and a `fold` function which combines the elements of the tree in a particular order. How can we define these functions?

[3]Indeed, "almost every" datatype admits a concept of mapping and folding. Proper treatment of this is outside the scope of this course, however.

Tree Mapping

The type signature for `treemap` will look similar to `map`:

`treemap : ('a -> 'b) -> 'a tree -> 'b tree`
REQUIRES: `f is total`
ENSURES: `treemap f T` evaluates to `T` with `f` called on each element

```
fun treemap f Empty = Empty
  | treemap f (Node (L, x, R)) =
      Node (treemap f L, f x, treemap f R)
```

Tree Folding

What about folding? We need to first pick a particular traversal order. Let's go with **inorder** traversal, which is the more intuitive "left-to-right" traversal.

```
treefoldl : ('a * 'b -> 'b) -> 'b -> 'a tree -> 'b
```
REQUIRES: `true`
ENSURES: `treefold f z T` $\cong$ `foldl f z (inord T)`

We see that the types of `foldl` and `treefoldl` look very similar.

Tree Folding

Now, let’s implement it.

```
fun treefoldl f z Empty = z
  | treefoldl f z (Node (L, x, R)) =
      let
        val left_folded = treefoldl f z L
      in
        treefoldl f (f (x, left_folded)) R
      end
```

Tree Folding: Example

Let’s see tree folding in action (click me!)

Tree Search

Using HOFs, we can also encapsulate the design pattern for a generic function which performs a search on trees.

To facilitate this, similarly to how `sort` took in a generic comparison function, our `search` function will take in an arbitrary **predicate** on elements, which returns a boolean on whether the element is what we are meant to search for.

```
search : ('a -> bool) -> 'a tree -> 'a option
```
REQUIRES: `p` is total
ENSURES: `search p T` evaluates to the first element in `T` that satisfies `p`, in inorder traversal

Tree Search

```
fun search p Empty = NONE
  | search p (Node (L, x, R)) =
    case search p L of
      NONE =>
        if p x then SOME x
        else search p R
    | SOME res => SOME res
```

This search function follows the same sort of logic as the original `inord` function we defined – it obeys the left-root-right ordering.

4 - Simplifying Programming

Towards Simpler Code

We've talked a lot about using HOFs to simplify our language. We obtain expressive functions from generalized templates of program logic, which is an improvement that is almost linguistic in nature.

It's not quite, since we're really writing code to write other code. But there are other useful applications of HOFs, which allows us to write literally simpler code.

For instance, suppose we had a very nested function application. It would look something like:

```
foo (bar (qux (baz x)))
```

The Pipe Operator

Such a nested expression is rather displeasing to the eye, as well as being generally annoying to deal with, due to the parentheses. It also requires being read "inside out", since `baz` comes after `foo`, but is evaluated first!

To that end, we can define the `|>`[4] operator, which allows us to *reverse* the order of function application. It is defined as:

```
infix |>

fun x |> f = f x
```

Thus, we could rewrite the first example as:

```
x |> baz |> qux |> bar |> foo
```

[4]Pronounced "pipe".

The Pipe Operator

This might seem weird to look at at first, but this is extremely useful for producing legible code, that reads like simple instructions.

For instance, take the following recipe:

- Heat oven to 400
- Insert tray of mozzarella sticks
- Wait two hours
- Remove charred remains

The Pipe Operator

Well, let's see what happens if we try to codify these instructions as actual SML code.

For instance, take the following recipe:

```
heat oven 400
|> insert trayOfMozzarellaSticks
|> wait 2
|> remove
```

A Student Problem

We can think of the pipe operator as stringing together operations, in the same way that we might construct a pipe out of components with compatible ends.
Let's take a look at another problem which might benefit from usage of the pipe operator. Suppose we would like to write:

`findStudentGrade : (string * string) -> string -> int`
REQUIRES: `true`
ENSURES: `findStudentGrade (student, assignment) file` looks up the grade of `student` on assignment `assigment` in the grades file `file`

And then, suppose we have the following helpers:

- `readFile : string -> string`, which reads in the contents of a file
- `parseGrades : string -> grades`, reads a string as a grade sheet
- `lookup : (string * string) -> grades -> int`, which tries to look up a student's grade on a given assignment

A Pipeable Problem?

It looks like we can just straightforwardly use `|>`!

```
fun findStudentGrade (student, assignment) grades_file =
  grades_file
  |> readFile
  |> parseGrades
  |> lookup (student, assignment)
```

Except, that might not actually be true. The pipe operator works because each of those operations can be applied as a function. But what if some of the functions we want to pipe are fallible? In other words, they return `t option`, for some type `t`?

Whatever Can Go Wrong...

We see that our helpers defined previously have a host of problems that might cause them to want to return `option` types!

- `readFile : string -> string option`, because the file might not exist
- `parseGrades : string -> grades option`, because the grades might not be in the correct format to be parsed
- `lookup : (string * string) -> grades -> int`, because the student might not exist

Let’s rewrite the code to accommodate this.

Reframed with Options

So let's try doing this example again, except now all our functions return options.

```
fun findStudentGrade (student, assignment) grades_file =
  case readFile grades_file of
    NONE => NONE
  | SOME file_content =>
    (case parseGrades file_content of
      NONE => NONE
    | SOME grades =>
      (case lookup (student, assignment) grades of
        NONE => NONE
      | SOME grade => SOME grade))
```

This is disgusting.

An Excess of Casing

In the previous example, we had to insert a `case` expression every single time that we wanted to unpack the result of a step. On some level, this is expected, since we otherwise have no way of dispatching on what exactly was returned, but the redundancy is in the `NONE` case.

When `SOME` is returned, we proceed as normal, but in every single case where we receive `NONE`, we just return `NONE`. This is *boilerplate logic*, because it just bloats the code, and doesn't add anything substantive to the interesting behavior of the function.

Fortunately, `option` is something called a **monad**.

Def A **monad** is a particular kind of type constructor that supports some operations that obey certain mathematical laws.

Enter the Monad

It's not actually super important what a monad is, but the main idea is that we can write a single function, `bind`:

```
fun bind (x : 'a option) (f : 'a -> 'b option) =
  case x of
    NONE => NONE
  | SOME res => f res
```

This kind of looks like the boilerplate logic around each one of our earlier steps!

What this function does is take a function which is supposed to operate on a non-optional value, which might fail. It then passes an optional value into it by handling the `NONE` case explicitly, as we did earlier. This takes care of the casing for us, so that we don't have to!

Piping with `bind`

Now, we can write:

```
fun findStudentGrade (student, assignment) grades_file =
  bind (readFile grades_file) (fn contents =>
  bind (parseGrades contents) (fn grades =>
  lookup (student, assignment) grades))
```

Now, much more readable!

Infix `bind`

Some enthusiasts also enjoy defining `bind` as an infix operator.

Traditionally, this is named `>>=`.[5]

They define:

```
infix >>=

fun x >>= f = bind (x, f)
```

to get:

```
fun findStudentGrade (student, assignment) grades_file =
  readFile grades_file >>= (fn contents =>
  parseGrades contents >>= (fn grades =>
  lookup (student, assignment)))
```

[5] This is big in the Haskell community.

Using operators like `>>=` and `|>`, which are really just higher-order functions, we can achieve vastly more readable code, which leverages simple principles to simplify program logic that would otherwise bloat a program.

It seems like a little, but code readability is really important when maintaining a codebase! We posited on the first day that functional programming is a refinement on our ability to communicate, and this is a concrete example of how small language features can provide almost linguistic benefits.

5 - Case Study: Staging (Bonus)

Benefits of Staging

Let's look at a concrete instance of a problem that could be staged, for performance benefits.

Consider a function `nth_largest`, which has the following specification:

```
nth_largest : int list -> int -> int
```
REQUIRES: `i >= 0`
ENSURES: `nth_largest L i` evaluates to the `i`th largest element in `L`

A Naive Approach

Here's a simple way that we might implement `nth_largest`. Assume that we have the `sort` function that we defined earlier.

```
fun nth_largest L i =
  let
    val sorted = sort (Int.compare, L)
  in
    List.nth (L, i)
  end
```

where `List.nth` is the function which gets the `n`th element of a list.

Simple enough!

Cost Analysis

But wait, what's the complexity of this function? We know that our previous work bound for sorting was $O(n \log n)$, when we implemented merge sort, and the standard library's `List.nth` function is $O(\texttt{i})$, where `i` is the index that we are trying to access.[6]

So then if we ran `nth_largest L i` on varying values of `i`, and we did it k times, we would incur $k \cdot n \log n$ in cost!

This is primarily because on *every call* to `nth_largest`, we sort the list. This is massively wasted effort! Can we do better?

[6] This is because `List.nth` just pops off the first `i` elements, until it reaches the `i`th one. There's no magic.

Doing Just Fine

We can do better.

We notice that there is no data dependency between sorting the list and the second argument, `i`. This means that the computation of `sort (Int.compare, L)` can be moved up!

```
fun nth_largest L =
  let
    val sorted = sort (Int.compare, L)
  in
    fn i => List.nth (sorted, i)
  end
```

A Staged Experience

Now, instead of calling `nth_largest L i` a bunch of times, we can first call `nth_largest L`, and obtain a *staged* version of `nth_largest`, which is particular to the list `L`. We can then query it as many times as we like, with only a linear cost each time.

```
val staged_nth_largest = nth_largest L (* O(n log n) *)

(* each of these are O(n) *)
val first_largest = staged_nth_largest 0
val second_largest = staged_nth_largest 1
val third_largest = staged_nth_largest 2
```

The Effect

Does it matter, at the end of the day? By sorting the list at all, we still incur a $O(n \log n)$ cost, so maybe we haven't actually saved much.

If we were only planning on doing a constant number of queries, it won't actually asymptotically matter, in the long run.

This particular situation isn't one that will be asymptotically improved, but it's still *better*. Avoiding a potentially large amount of duplicate sorts is still a big deal. It's important to be mindful of the computations you're doing, and see opportunities for improvement.

Thank you!