Contents

1 Higher Order Functions (A Course Numbering Problem)

1.1 Currying

Previously, if we've wanted a function to take in multiple arguments, we've passed in a tuple of those arguments.

With **curried functions**, we pass in one argument, and the function application evaluates to *another* function that takes in the remaining arguments.

Example(s):

1.2 A Map

```
(* map :                                                        *)
fun map
```

Example(s):

1.3 Folds

```
(* foldr :                                                  *)
fun foldr
```

```
(* foldl :                                                  *)
fun foldl
```

Example(s):

1.4 Compose

```
(* op o :                                                   *)
```

Example(s):

2 Adding the Curry

In `addmul.sml`, we have defined the following functions:

```
fun incr (y : int) : int = 1 + y

fun addTwo (x : int) (y : int) : int = x + y
```

Task 2.1.

Which of the following are correct ways to write the type of `addTwo`?
Hint: `->` is **right-associative**.

1. `int -> (int -> int)`
2. `(int -> int) -> int`
3. `int -> int -> int`

Solution 1.

1 and 3 are correct, and are equivalent.

The following tasks should be done in `addmul.sml`.

Task 2.2.

Define the function

`incr1 : int -> int`

REQUIRES: true

ENSURES: `incr1` $\cong$ `incr`

Constraint: Declare your function using `val`, and use `addTwo`.

Solution 2.

```
val incr1 = addTwo 1
```

Task 2.3. (Recommended)

Define the function

`add3 : int -> int -> int -> int`

REQUIRES: true

ENSURES: `add3 x y z` $\cong$ `x + y + z`

Note that `add3 x y z` $\cong$ `((add3 x) y) z`, because function application is **left-associative**.

Constraint: Declare your function using `fun`.

Solution 3.

```
fun add3 x y z =
  x + y + z
```

Task 2.4.

Define the function

```
mul3 : int -> int -> int -> int
```

REQUIRES: true

ENSURES: `mul3 x y z` $\cong$ `x * y * z`

Constraint: Declare your function using `val`.

Solution 4.

```
val mul3 = fn x => fn y => fn z =>
  x * y * z
```

3 3…2…1… Blast-HOF!

3.1 Zip it, it's time to Filter

Let's practice implementing some more higher order functions!

For each of the following tasks, write the function in `code/implementing-hofs/implementing-hofs.sml`.

Task 3.1. (Recommended)

`filter : ('a -> bool) -> 'a list -> 'a list`

REQUIRES: `p x` is valuable for all `x` in `L`

ENSURES: Returns the list of those `x` in `L` for which `p x` $\Longrightarrow$ `true`

Solution 1.

```
fun filter p [] = []
  | filter p (x::xs) =
      if p x then x::filter p xs else filter p xs
```

Task 3.2. (Recommended)

Now, implement the function

`zipWith : ('a * 'b -> 'c) -> 'a list * 'b list -> 'c list`

REQUIRES: true

ENSURES: `zipWith f (L1,L2)` $\Longrightarrow$ `L'`, where
`L'` $\cong$ `map f (ListPair.zip (L1,L2))`

Note that `ListPair.zip (L1,L2) : 'a list * 'b list -> ('a * 'b) list` takes two lists and "zips" them together by pairing them up element by element until either list runs out of elements. Some examples:
`ListPair.zip ([1,2,3],["a","b","c"])`$\cong$ `[(1,"a"),(2,"b"),(3,"c")]`
`ListPair.zip ([1,2,3,4],["a","b"])`$\cong$ `[(1,"a"),(2,"b")]`

Constraint: You may not use `map` or `ListPair.zip` in your implementation. Do this recursively!

Solution 2.

```
fun zipWith f ([],_) = []
  | zipWith f (_,[]) = []
  | zipWith f (x::xs,y::ys) = f (x,y) :: zipWith f (xs,ys)
```

In the next task, you will be writing the same functions twice: once recursively without any higher-order functions, and once using HOFs.

Task 3.3.

Implement the function

> `mapPartial : ('a -> 'b option) -> 'a list -> 'b list`
>
> REQUIRES: true
>
> ENSURES: `mapPartial f L` $\Longrightarrow$ `L'` where `L'` contains all elements `y` such that for `x` in `L`, `f x` $\cong$ `SOME y`

Constraint: For `mapPartial`, you should write this function recursively, without using any built-in HOFs. For `mapPartial'`, you should use higher-order functions, but you may not use `filter` or `map` in your solution.

Solution 3.

```
fun mapPartial f [] = []
  | mapPartial f (x::xs) = (
      case f x of
        SOME z => z :: mapPartial f xs
      | NONE   => mapPartial f xs
    )
```

```
(* EXPLANATION: List.foldr takes in elements of L from right side,
 * checking 1 element (x) at a time - if `f x` evaluates to SOME y,
 * then the result (y) should be added to the accumulator list.
 * NOTE, foldr is used to preserve list order! *)
fun mapPartial' f L =
  List.foldr (fn (x,acc) =>
    case f x of
      SOME y => y::acc
    | NONE   => acc
  ) [] L
```

4 Point-Free Programming

You might find the built-in infix composition function `o` handy, as well as the built-in List library.

It's fine if the functions you define have more general types than the ones listed below. Ignore the value restriction if you run into it.

For each of the following tasks, write the function in `code/pointfree/pointfree.sml`.

First, we are going to implement some functions without the constraint of being point-free (to better prepare you for the point-free versions). We will, however, have the following constraint:

Constraint: Define the functions without using `fun`.

Task 4.1.

`sum_with_lambda : int -> int list -> int`

REQUIRES: true

ENSURES: `sum_with_lambda n l` sums the elements of `l`, adding `n` to the sum

Solution 1.

```
val sum_with_lambda = fn n => fn L => foldr (op+) n L
```

Task 4.2.

`sum_both_lambda : int list -> int list -> int`

REQUIRES: true

ENSURES: `sum_both_lambda l1 l2` sums the elements of `l1` and `l2`

Solution 2.

```
val sum_both_lambda = fn L1 => fn L2 => foldr (op+) (foldr (op+) 0 L1) L2
```

Now, we are going to implement the same functions except with the following constraint:

Constraint: Define the functions without using `fun` or `fn`.

Task 4.3.

`sum_with : int -> int list -> int`

REQUIRES: true

ENSURES: `sum_with n l` sums the elements of `l`, adding `n` to the sum

Solution 3.

```
val sum_with = foldr (op +)
```

Task 4.4. (Recommended)

`sum_with' : int * int list -> int`

REQUIRES: true

ENSURES: `sum_with' (n,l)` sums the elements of `l`, adding `n` to the sum

Solution 4.

```
val sum_with' = foldr (op +) 0 o (op ::)
```

Task 4.5.

`sum_both : int list -> int list -> int`

REQUIRES: true

ENSURES: `sum_both l1 l2` sums the elements of `l1` and `l2`

Solution 5.

```
(* EXPLANATION: sums the ints of the first list with foldr
 * and uses the result as the starting accumulator to the 2nd foldr call,
 * which sums the ints of the second list *)
val sum_both = foldr (op +) o foldr (op +) 0
```

Task 4.6.

`sum_both' : int list * int list -> int`

REQUIRES: true

ENSURES: `sum_both' (l1,l2)` sums the elements of `l1` and `l2`

Solution 6.

```
(* EXPLANATION: appends the lists together and then sums the ints of the
    resulting list with foldr *)
```

```
val sum_both' = foldr (op +) 0 o (op @)
```

5 Don't Nod Off, There's More HoF

Recall the definitions of `foldl` and `foldr`:

```
fun foldl (cmb : 'a * 'b -> 'b) (z : 'b) (L : 'a list) : 'b =
  case L of
    [] => z
  | x :: xs => foldl cmb (cmb (x, z)) xs

fun foldr (cmb : 'a * 'b -> 'b) (z : 'b) (L : 'a list) : 'b =
  case L of
    [] => z
  | x :: xs => cmb (x, foldr cmb z xs)
```

Consider that `fold`ing generalizes the idea behind many of the functions we've written in 15-150 up until now: write a base case (`z`), and then building up a return value by accumulating the result of applying some part of the value to (`cmb`). Let's prove how true this is by rewriting some familiar functions using only `foldl`/`foldr`!

For each of the following tasks, write the function in `code/using-hofs/use-hofs.sml`.

Task 5.1.

Consider

```
fun sum (L : int list) : int =
  case L of
    [] => 0
  | x :: xs => x + sum xs
```

Rewrite `sum` using `foldl`/`foldr`.

Solution 1.

```
val sum = foldr op+ 0
```

Task 5.2. (Recommended)

Consider

```
fun rev ([] : int list) : int list = []
  | rev (x::xs) = (rev xs) @ [x]
```

Rewrite `rev` using `foldl`/`foldr`.

Solution 2.

```
fun rev L = foldl op:: [] L
```

Task 5.3. (Recommended)

Consider

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

Rewrite `flatten` using `foldl`/`foldr`.

Solution 3.

```
fun flatten L = foldr op@ [] L
```

Before we move on to more exciting functions we can implement using HOFs, let's take some time to consider how the types of these HOFs (and their inputs) are affected when given an input with a particular type.

Task 5.4.

Consider

```
val boolify = foldl g true [1, 2, 3]
```

What is the type of `g`?

Solution 4.

The type of `g` is `int * bool -> bool`, since the second input to `foldl` has type `bool` and the third input has type `int list`.

For the following tasks, consider

```
val toStringify = foldr (fn (x, y) => (Int.toString x) ^ y) z
```

Task 5.5.

What is the type of `z`?

Solution 5.

The type of `z` is `string`, since the first input to `foldr` has type `int * string -> string`.

Task 5.6.

What is the type of the function `toStringify`?

Solution 6.

The type of `toStringify` is `int list -> string`, since the type of the first input to `foldr` is `int * string -> string`, indicating that the third input should have type `int list` and the output should have type `string`.

For the following tasks, consider the datatype `'a idxTree`, defined as

```
datatype 'a idxTree = Empty'
                    | Node' of 'a idxTree * (int * 'a) * 'a idxTree
```

Task 5.7.

Consider

```
val treeify = foldl g (Node'(Empty',(0, 0),Empty')) ["15","1","50"]
```

What is the type of `g`?

Solution 7.

The type of `g` is `string * int idxTree -> int idxTree`, since the second input to `foldl` has type `int idxTree` and the third input to `foldl` has type `string list`.

Task 5.8.

Now, suppose we had

```
val treeify' = foldl g Empty' [1, 2, 3]
```

Give a type for the function `g` such that the declaration is well-typed.

Solution 8.

The type of `g` can be any instance of the type `int * 'a idxTree -> 'a idxTree` (including `int * 'a idxTree -> 'a idxTree` itself) since there are no constraints on what the type of `Empty'` should be other than `'a idxTree`.

Task 5.9.

Consider the following code

```
fun inord Empty' = []
  | inord (Node'(L, v, R)) = inord L @ (v :: inord R)

fun treeFold g z T = foldr g z (inord T)
```

What is the type of the function `treeFold`?

Solution 9.

The type of `treeFold` is `((int * 'a) * 'b -> 'b) -> 'b -> 'a idxTree -> 'b`.

First, we note that the type of `inord` is `'a idxTree -> (int * a) list`, which means `inord T` has type `(int * 'a) list` for all `T : 'a idxTree`. Since `inord T` has type `(int * 'a) list`, it must be that the function `g` has type `(int * 'a) * 'b -> 'b`, since the type of the elements in the list being folded over is `int * a`. Finally, we get that `z : 'b` from the type of `g`, which gives us our overall type.

Now that we've tried rewriting simple functions using `foldl`/`foldr` and have considered the types of expressions with HOFs, it's time for us to try something more interesting using all the HoFs we've seen so far!

For each of the following tasks, write the function in `code/using-hofs/use-hofs.sml`.

Task 5.10.

```
maxBy : ('a * 'a -> order) -> 'a list -> 'a option
```

REQUIRES: `cmp` is total

ENSURES:

$$\texttt{maxBy cmp L} \Longrightarrow \begin{cases} \texttt{SOME x} & \text{where } \texttt{x} \text{ is the maximum element in } \texttt{L} \text{ according to } \texttt{cmp} \\ \texttt{NONE} & \text{if the list is empty} \end{cases}$$

Solution 10.

```
fun maxBy cmp L =
  let
    fun cmp' x (SOME acc) = (case cmp (x, acc) of
                                GREATER => SOME x
                              | _       => SOME acc)
      | cmp' x  NONE      = SOME x
  in
    foldr (Fn.uncurry cmp') NONE L
  end
```

Task 5.11.

```
gradebook : int list list -> int list -> int list
```

REQUIRES: For all `S` in `scores`, $|\texttt{S}| = |\texttt{weights}|$

ENSURES: `gradebook scores weights` returns a list `L` of the same length as `scores` such that

$$\texttt{L}[i] \cong \sum_{j=0}^{|\texttt{S}|-1} \texttt{scores}[i][j] * \texttt{weights}[j]$$

Note: Which HoF can we use to combine two lists?

Example:

```
val scores = [[10, 10, 10], [9, 10, 8], [9, 9, 9], [5, 10, 10]]
val weights = [10, 10, 20]
val [400, 350, 360, 350] = gradebook scores weights
```

Solution 11.

```
fun zipWith _ _        []        = []
  | zipWith _ []       _         = []
  | zipWith f (x::xs) (y::ys) = f (x, y) :: zipWith f xs ys
val sum = foldl op+ 0
fun gradebook scores weights =
  map (fn L => (sum o zipWith op* L) weights) scores
```

6 The three best things in life: Money, Pipes, and Curry

The following tasks should be done in `combinators.sml`.

6.1 Apply

Consider the following: You start out with some piece of data `x : t1`. You first want to transform it into something else using a function `f1 : t1 -> t2`. Then you want to transform that result with a function `f2 : t2 -> t3`. And so on.

An expression like this will do the trick:

```
f8 (f7 (f6 (f5 (f4 (f3 (f2 (f1 x)))))))
```

There's problems with this, however.

- There's a lot of parentheses.
- Everything is written "backwards." That is, the original piece of data that we start with is written *after* the function that does the first transformation, which is written *after* the function that does the second transformation, and so on.

Let's solve the first problem with an infix operator `<|`, pronounced "apply." Such a `<|` would be defined like this:

```
infixr <|
fun L <| R = ???
```

We can then use it like this:

```
f8 <| f7 <| f6 <| f5 <| f4 <| f3 <| f2 <| f1 <| x
```

Because we said `<|` is a **right-associative infix** operator (hence `infixr`), everything will be done in the correct order.

Task 6.1.

What is the type of `<|`?

Solution 1.

```
op <| : ('a -> 'b) * 'a -> 'b
```

Task 6.2.

Define `<|`.

Solution 2.

```
fun f <| x =
  f x
```

Note that in haskell, this operator is the `$` function.

6.2 Hype for Pipes

We fixed the problem of lots of parentheses. But everything's still in the wrong order.

Let's define a new infix operator `|>`, pronounced "pipe." We will be able to use it like this:

```
x |> f1 |> f2 |> f3 |> f4 |> f5 |> f6 |> f7 |> f8
```

Such a `|>` would be defined like this:

```
infix |>
fun L |> R = ???
```

Once you figure our the definition of `|>`, feel free to paste it everywhere in your SML files and call all your functions with it. Doesn't it read nicely!?!?

Task 6.3.

Notice that this time, we said `infix`, not `infixr`. This means `|>` is **left-associative**. Why does this make sense?

Solution 3.

Because we go from left to right to do the function applications.

Task 6.4.

What is the type of `|>`?

Solution 4.

```
op |> : 'a * ('a -> 'b) -> 'b
```

Task 6.5.

Define `|>`.

Solution 5.

```
fun x |> f =
  f x
```

6.3 Curry

Task 6.6. (Recommended)

Define the function

```
curry : ('a * 'b -> 'c) -> ('a -> 'b -> 'c)
```

REQUIRES: true

ENSURES: `f (x, y)` ≅ `curry f x y`

Note that we could have written the right hand side as `(curry f) x y`.

Hint: `curry` takes in 3 arguments:

- An uncurried function `f : ('a * 'b -> 'c)`,
- A value `x : 'a`,
- A value `y : 'b`.

Follow the types!

Solution 6.

```
(* curry : ('a * 'b -> 'c) -> ('a -> 'b -> 'c) *)
fun curry f =
  fn x => fn y => f (x, y)
```

```
(* curry : ('a * 'b -> 'c) -> ('a -> 'b -> 'c) *)
fun curry f x y = f (x, y)
```

Task 6.7. (Recommended)

Define the function

`uncurry : ('a -> 'b -> 'c) -> ('a * 'b -> 'c)`

REQUIRES: true

ENSURES: `f x y` ≅ `uncurry f (x,y)`

Solution 7.

```
fun uncurry f (x, y) =
  f x y
```

7 Folding is Entirely Overpowered

Given how general the concept of `folding` is, might it be possible to write other HOFs in terms of `fold`? In this task, we'll see that, in fact, it is!

Task 7.1.

Consider

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

Define `map_cmb` and `map_z` such that

- for all types `t1`,
- for all types `t2`,
- for all values `f : t1 -> t2`,

we have that

`foldr (map_cmb f) map_z` $\cong$ `map f`

Solution 1.

```
fun map_cmb f (x, ac) =
  f x :: ac
val map_z =
  []
```

Task 7.2.

Consider

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

Define `filter_cmb` and `filter_z` such that

- for all types `t`,
- for all values `p : t -> bool`,

we have that

`foldr (filter_cmb p) filter_z` $\cong$ `filter p`

Solution 2.

```
fun filter_cmb p (x, ac) =
  if p x
  then x :: ac
  else ac
val filter_z =
  []
```

Task 7.3.

Warning: this is quite tricky. Don't worry if you can't get it!

Define `foldl_cmb` and `foldl_z` such that

- for all types `t1`,
- for all types `t2`,
- for all types `t3`,
- for all values `cmb : t1 * t2 -> t2`,
- for all values `z : t2`,
- for all values `xs : t1 list`,

we have that

`foldr (foldl_cmb cmb) foldl_z xs z` $\cong$ `foldl cmb z xs`

Solution 3.

```
fun foldl_cmb cmb (x, ac) y =
  ac (cmb (x, y))
fun foldl_z x =
  x
```

Task 7.4.

Explain how you came to your answer to the previous task and why it works.

Solution 4.

This is quite daunting, but we can make it less daunting by making some observations:

- If we invent `F` so that `foldr (foldl_cmb cmb) foldl_z` $\cong$ `F`, then we know `F []` $\cong$ `foldl_z`, and that `F (x::xs)` $\cong$ `(foldl_cmb cmb) (x, F xs)`

- This allows us to think about `foldl_cmb cmb` and `foldl_z` more concretely. `foldl_z` is the base case of `F` and `foldl_cmb cmb` tells `F` what to do with `x` and the output of a recursive call to `F` on `xs`.
- This lets us simplify the type we must think about by considering the type of `F` in the context of the problem, which is `t1 list -> (t2 -> t2)`

So let's try to write a function `F : t1 list -> (t2 -> t2)` such that

`F xs z` $\cong$ `foldl cmb z xs`

.

Conceptually, we need to write a function that takes in a list, and gives back a function `f` so that when a base case is passed in to `f`, `f` accumulates the base case down the list from left to right. Remember how in the implementation of `foldl`, the base case is used as an accumulator argument? Well, since now the base case gets passed into the ***output of*** `F`, we will need to treat the ***input of the output of*** `F` as an accumulator argument!

How does this look? Well, for the base case of `F`, `F []` should return a function so that passing in some `z` yields the same result as folding left with `[]` and `z`. But this result is just `z`! Therefore, we want `F [] = fn z => z`.

In the recursive case, we get to assume that `F xs` gives us a function which when passed a `z`, folds `z` down `xs` using `cmb`. We must provide a function which takes in a base case `z1`, and folds `z1` down the list `x::xs` using `cmb`. We can do this by first computing `cmb(x, z1)`, and then sending this result as the new base case value to the input of a recursive call! This is exactly analogous to how `foldl` first computes the folding function and then sends it as an argument to a recursive call. the actual code looks like this:

```
fun F [] = fn x => x
  | F (x::xs) = fn z1 => (F xs) (cmb(x, z1))
```

Now that we have `F`, we can figure out the construction of `foldl_cmb` and `foldl_z`. `foldl_z` is the base case of `F`. `foldl_cmb` is written in terms of `cmb`, and tells `F` what to do with `x` and `F xs` by taking them in as arguments as a tuple `(x, ac)`, (where `ac` is the same thing as `F xs`). Thus, we can say:

```
fun foldl_z z = z

fun foldl_cmb cmb = fn (x, ac) => fn z1 => ac (cmb(x, z1))
```

A fun note about this problem: Notice how in the recursive case of `F`, we have `(F xs) ( cmb(x, z1))`. This looks extreemely similar to `foldl` if you remove the parenthesis! This changing of the order is exactly what allows us to implement `foldl` with `foldr`. `foldl` requires an accumulator argument, but putting that argument as an input to the recursive function means doing the recursion ***after*** the evaluation of the folding function. On the other hand, `foldr` requires the recursion to be done ***before*** the folding function. It is only by making the accumulator argument an input to the ***output*** of the recursive function, that we are able to acumulate by ***first*** recursing, ***and then*** evaluating the folding function, which is exactly what lets us implement `foldl` using `foldr`!