S1.3: Higher Order Functions

CSci 2041:

Advanced Programming Principles

University of Minnesota,
Prof. Van Wyk,
Spring 2022

Values

So now, we've seen 3 varieties of values (and types):

1. primitive values and types: `int`, `bool`, etc
   These are simple and easy to understand.

2. lists and tuples:
   e.g. `string list`, `int * string`

3. functional values and types:
   ```
   let inc x = x + 1
   inc: int -> int
   ```


Functional values

We now turn our attention to functions.

Specifically, languages in which functions are "first class citizens." They are "just values."

They can be

- defined and associated with a name
  (typical `let` expressions)
- passed as input to other functions
- returned as values from other functions
- specified as literal values that are not given a name (lambda expressions)

Our big questions are:

- How can we structure computations in such languages?
- How can code be easily reused?
  This is one of our goals.

Topics

These slides cover the following topics:

- passing "helper functions" as arguments

  For example, consider a `find_by` function with the type

  e.g. `find_all_by : ('a ->bool) -> 'a list -> 'a list`

  that uses a helper function the check for equality when checking if an element appears in a list.
- specifying functional values using lambda expressions and curried functions
- higher order functions embodying computational patterns, for example

  `map: ('a -> 'b) -> 'a list -> 'b list`

  `fold: ('b -> 'a -> 'b) -> 'b -> 'a list -> 'b`

Functions needing a form of equality check

Many functions over lists require a check for some form of equality.

For example

- is-element-of, lookup
- grouping, partitioning
- splitting at a certain value

Let's consider the `lookup_all` function from S1.2 and how we can use a more general (in some respects) function named `find_all_by`.

We can then use `find_all_by` in another case to implement a is-element-of function.

See examples in `find_and_lookup.ml` in the `Sample-Programs` directory.

Can we write `find_all_by` ?

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


Other examples

There are other circumstances in which we may want to specify the function used for checking for some notion of equality:

- list membership, `is_elem`
- functions to group or partition a list of values
- set functions:
  - `union, intersect, setMinus, nub`
- binary tree `insert` function

Specifying these "helper" functions

- Functions like `find_all_by` need to be passed some sort of equality-like checking function.
- How can we specify these?
  The specification of `streq` in the examples was somewhat cumbersome.
- We have a few options
  - `let`-declared functions
  - lambda expressions
  - converting operators into functions
  - use of curried functions.

Lambda Expressions

- Lambda expressions let use write function values directly.
- (Historically, these are written as $\lambda x \rightarrow x + 1$, as part of Alonzo Church's "lambda calculus" for studying theoretical ideas in computation.)
- This is similar to writing integer, string, or list values directly without the need to give them a name.
  e.g. `1`, or `[3.4; 5.6; 7.8 ]`
- Lambda expressions in OCaml: `fun` *formal parameters* `->` *body*
  We often wrap these in parenthesis to group things properly.
- *e.g.*
  - `fun x y -> x = y`
  - `fun x -> x + 1`
- Use `find_all_by` to find all even numbers in a list or the numbers larger than some threshold.

Converting operators into functions

- OCaml allows many infix operators to be used as functions by wrapping them in parenthesis.
  - `(+) : int -> int -> int`
  - `(=) : 'a -> 'a -> bool`
  - `(<) : 'a -> 'a -> bool`
  - However, `::` is not treated this way. So `(::)` does not work.

    `::` is a value constructor, not an operator or function.
- Let's define an equality function for `find_all`.
- Let's rewrite `big_nums` to use this technique.

Use of curried functions

Recall the type of `find_all_by`

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

We could define a "default" find function as follows:

- `let find_all x xs = find_all_by ((=) x) xs`

What is the type of `find_all`?

Exercise S1.3: #1: Another `find_all`

We saw that we could define a "default" find function as follows:

- `let find_all x xs = find_all_by ((=) x) xs`

Is there a shorter definition of `find_all`?

Exercise S1.3: #2: Finding long strings

Write a function named `big_strs` which takes a integer `n` and a list of strings and returns the list containing all strings whose length is strictly bigger than `n`.

Your function must call `find_all_by`, repeated below.

```
let rec find_all_by f l =
  match l with
  | [] -> []
  | x::xs ->
     let rest = find_all_by f xs
     in if f x then x::rest else rest
```

Let's work on this for 2 minutes.

"partial application"

The term "partial application" is not technically correct for a language like OCaml with curried functions.

With curried functions, the function type explicitly indicates that arguments are passed in one at a time.

- `add: int -> (int -> int)`

Function application only takes one operation at a time.

- `add 3 4` is the same as `(add 3) 4`.
- (The parenthesis are not required in the type or the application of `add`.)

But these work seamlessly together so that it may feel like we are passing in more than one argument at once even though the mechanisms implementing functions don't work that way.

"partial application"

If C allowed partial application, then for a function like `add`

- `int add (int x, int y) { return x + y; }`

then "partial application" might look like

- `add ( 3, _ )`

and evaluate to a function that takes and integer and returns an integer.

But of course this isn't possible in C.

The point is that "partial application" is not needed in a language with curried functions.

Ordering functions

There are also many examples of computations that require ordering values as *equal*,*less than*, or *greater than*.

- sort a list given an ordering function
  `List.sort: ('a -> 'a -> int) -> 'a list -> 'a list`
- merge two sorted lists
  `List.merge: ('a -> 'a -> int) -> 'a list -> 'a list -> 'a list`
- split a list into three
  `splitOnCompare: ('a -> 'a -> int) -> 'a -> 'a list -> ('a list, 'a list, 'a l`
- `min: ('a -> 'a -> int) -> 'a list -> 'a option`
- `max: ('a -> 'a -> int) -> 'a list -> 'a option`

See `compare: 'a -> 'a -> int`.

Exercise S1.3: #3: Write a version of `List.merge`

Let's write our own version of 'merge'.

It should have the type:
`merge: ('a -> 'a -> int) -> 'a list -> 'a list -> 'a list`

Additional examples

Functions drop_while, drop_until:

- These have the type
  `('a -> bool) -> 'a list -> 'a list`
- They return some portion of the original list, after dropping all items that return true (or false) when provided to the function.

Functions `take_while` and `take_until` are similar.

Exercise S1.3: #4: Write `drop_while`

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


More functions over functions

We can easily write functions

- change the order of arguments in a function
- compose two functions
- "curry" or "uncurry" a function

Consider `flip`:

```
let flip f a b = f b a
```

Let's revisit the problem with `big_nums` and `((>) n)` we saw earlier.

Exercise S1.3: #5: Function composition

Write a function to compose two functions.

Your function should be named `compose` and the type

`compose : ('b -> 'c) -> ('a -> 'b) -> 'a -> 'c`.

Finish this:

```
let compose f g x = ...
```


So far ...

- We've seen how to pass "helper" equality or ordering functions into list and tree processing functions such as `find_all_by` and `sort`.
- We've seen how to specify functions in a number of ways:
  - let-expr declarations
  - lambda expressions
  - using curried functions
  - converting operators into functions
- We now consider functions that implement different "design patterns" of computations over lists.

Map, Filter, and Fold

We can use higher order functions to perform computations over lists where we might otherwise write a recursive function.

For example,

```
let inc x = x + 1
let r1 = map inc [1;2;3;4;5]

let even n = n mod 2 = 0
let evens = filter even [1;2;3;4;5;6;7]

let sum xs = fold (+) 0 xs
```

The concepts of map, filter, and various folds are common in functional languages and their libraries.

We'll define our own implementations and later compare them to some standard library implementations.

Map

It is common to need to apply a function to every individual element of a list, returning a list with the results of those applications. For example,

```
let inc x = x + 1
let r1 = map inc [1;2;3;4;5]

let r2 = map int_of_char [ 'a'; '^'; '4' ]

let r3 = map Char.lowercase [
   'H'; 'e'; 'l'; 'l'; 'o'; ' '; 'W'; 'O'; 'R'; 'L'; 'D' ]
```

We can try these with `List.map` in utop first.

See examples in the utop-histories of use of `map` in `higher.ml`.

Exercise S1.3: #6: What is the type of `map`?

Recall our examples:

```
map inc [1; 2; 3; 4]
```

or

```
map Char.code [ 'a'; '@'; '4' ]
```

Exercise S1.3: #7: What is an OCaml implementation of `map`?

Recall our examples:

```
map inc [1; 2; 3; 4]
```

or

```
map Char.code [ 'a'; '@'; '4' ]
```


Parametric polymorphism

The importance of parametric polymorphism is hard to understate here:

The type of `map` is
`('a -> 'b) -> 'a list -> 'b list`.

Without this kind of polymorphism, we would be left writing individual functions for each type:

- `map_int_int: (int -> int) -> int list -> int list`
- `map_int_char: (int -> char) -> int list -> char list`
- ...

Lambda expressions are commonly used with applications of `map`.

Why write

```
let inc x = x + 1

  ... map inc [1;2;3;4;5] ...
```

when you could just write
`... map (fun x -> x + 1) [1;2;3;4;5] ...`

over strings

There are a number of simple examples of higher order functions that work over strings, when strings are lists of characters.

But the OCaml type `string` is a built-in type.

We'll define our own string type:

```
type estring = char list
```


Some sample functions over strings:

- `get_excited : estring -> estring`
  Convert all periods to exclamation marks (bangs) !
- `chill : estring -> estring`
  Convert bangs to periods.
- `freshman: estring -> estring`
  Convert all periods and bangs to question marks.

See examples in `higher.ml` in the code examples directory of the pubic repository.

Filtering elements from a list

It is also common to filter some elements from a list.

```
let even n = n mod 2 = 0
let evens = filter even [1;2;3;4;5;6;7]

let positive x = x > 0.0
let pos_nums = filter positive
                 [1.2; 3.4; -5.6; -7.8; 9.0]

let is_blank_or_tab ch = ch = ' ' || ch = '\t'
let ws = filter is_blank_or_tab (explode "a b\t c d")
```

See examples in `higher.ml`

Exercise S1.3: #8: What is the type of `filter`?

Recall our examples:
`filter even [1;2;3;4;5;6;7]`
or
`filter positive [1.2; 3.4; -5.6; -7.8; 9.0 ]`

Exercise S1.3: #9: What is an OCaml implementation of `filter`?

Recall our examples:
`filter even [1;2;3;4;5;6;7]`
or
`filter positive [1.2; 3.4; -5.6; -7.8; 9.0 ]`

- Let's consider filters over strings and revisit `higher.ml`
- Perhaps a function, `smush`, that removes all whitespace.
- Or a function to remove all punctuation. We will choose to disregard punctuation in our paradelle program, so this might be useful.

Exercise S1.3: #10:

Write a function that returns its input `char list` after removing all upper case letters from it.

(Well, lets just consider `'A'`, `'B'`, `'C'`, and `'D'` to keep this simple.)

And do it without using an `if-then-else` expression. Use `match`.

And, of course, use `filter`.

Folding lists

Another common idiom is to "fold" list elements up into a, typically, single value.

```
let a_sum = fold (+) 0 [1;2;3;4]
```

```
let sum xs = fold (+) 0 xs
```


Exercise S1.3: #11: What is the type of `fold`?

Recall our example:

```
fold (+) 0 [1; 2; 3; 4]
```

and perhaps

```
fold (+.) 0.0 [1.0; 2.0; 3.0; 4.0]
```

30 seconds - don't think too hard - just write.

Exercise S1.3: #12: What is OCaml implementation of `fold`?

Recall our example:
`fold (+) 0 [1;2;3;4]`

We can see this as
`1 + (2 + (3 + (4 + 0)))`

(1 minute - just sketch something.)

Exercise S1.3: #13: What is OCaml implementation of `fold`?

Or when we see
`fold (+) 0 [1;2;3;4]`

as
`((((0 + 1) + 2) + 3) + 4)`

(1 minute - just sketch something.)

Folding from the left or the right

Folding from the left, we first apply `f` to the first element `x` and the accumulator `accum` and this result is passed in as the accumulator for the next step.

```
let rec foldl (f: 'b -> 'a -> 'b) (accum: 'b) (lst: 'a list) : 'b =
  match lst with
  | [] -> accum
  | x::xs -> foldl f (f accum x) xs
```

Folding from the right, we apply `f` to the first element `x` and the result of folding up the rest of the list.

```
let rec foldr (f: 'a -> 'b -> 'b) (lst: 'a list) (base: 'b) : 'b =
  match lst with
  | [] -> base
  | x::xs -> f x (foldr f xs base)
```

Some more examples

- `length: 'a list -> int`
- `and: bool list -> bool`, also `or`
- `max: int list -> int option`, also `min`
- `is_elem: 'a -> 'a list -> bool`
- `split_by: 'a list -> 'a list -> 'a list list`
- `lebowski: char list -> char list`
  replace all `'.'` with
  `['.'; ' '; 'd'; 'u'; 'd'; 'e'; '.']`

Let's write some of these, both as recursive functions and usign `foldl` or `foldr`.

We'll ask: Is `foldl` or `foldr` better for any of these? Why?

These are found in `higher.ml` in the code examples directory in the public course repository.

The types

Look at the code and examples and consider the types of these functions.

Consider how `foldr` is just a "homomorphsim" from lists to the type being returned. (wait, what?)

Exercise S1.3: #14: Partitioning

Consider writing a function that partitions a list into two sub-lists based on a predicate?

```
partition: ('a -> bool) -> 'a list -> 'a list * 'a list
```

```
partition even [1;2;3;4;5;6;7;8;9;10]
```

evaluates to

```
( [2; 4; 6; 8; 10], [1; 3; 5; 7; 9] )
```

- what is the base value?
- what must the folding function do?

Exercise S1.3: #15: Grouping by 3

Can we write a function that groups a list of elements into a list of sub-lists of length 3 (or fewer, for the last sub-list).

```
group_by_3 [1;2;3;4;5;6;7;8;9;10]
```

evaluates to

```
[[1; 2; 3]; [4; 5; 6]; [7; 8; 9]; [10]]
```


Technique: accumulating parameters as "state" in folds

Consider the recursive function `even_length`:

```
let rec even_length (lst: 'a list) : bool = match lst with
  | [] -> true
  | [ _ ] -> false
  | _ :: _ :: rest -> even_length rest
```

- How can we write this as a fold since `fold_left` and `fold_right` both consume list elements one at a time?
- Use the folded-up value to maintain some form of state.
- Here, this state will indicate if list seen so far is even in its length.
- Let's do this using `fold_left`, call it `even_length_fl`.

Exercise S1.3: #16: Now write `even_length_fr`

Can you now implement this using `fold_right`?

Recall

```
let rec even_length (lst: 'a list) : bool = match lst with
  | [] -> true
  | [ _ ] -> false
  | _ :: _ :: rest -> even_length rest
```

Technique: extracting final answer from accumulating parameters

When the accumulator isn't the answer, but the answer is some component of the accumulator.

For example, how could we add up all the `int` values in a list that are in even positions in the list?

- `sum_even_positions [10;2;30;4] = 40`
- `sum_even_positions [10;2] = 10`
- `sum_even_positions [] = 0`

We may need to do some work on the result of a fold to compute the answer.

Here, the answer might be a value inside a pair that is the result of the fold.

Some additional problems

These, in `higher.ml`, are ones where a fold is used, but

- maybe not for all values of the input (`product_excp`)
- we need a notion of "local state" that keeps track of where the fold is in the processing of the list (`even_length_fl`, `even_length_fr`)
- the result coming out of the fold must be processed further to compute the final desired result. (`sum_even_positions`)

We look at all these different examples because they contain patterns and ideas that you may need to use in programming assignments.

In fact, you should come back to these examples when doing the homework and ask yourself which examples are similar to homework problems and what ideas from the examples can be used in the homework.

Seeing map, filter, fold as loops

It may be helpful to initially think of imperative solutions and consider what *state* is updated each time through the loop?

What is the style of loop that would implement

- map
- filter
- fold ?

For example, folds

In C:

```
sum = 0;
for (i=0; i<N; i++) {
  sum = sum + array[i];
}
```

The "accumulator" value in a fold is this state.

Of course, `i` is just an index so we don't see it in a functional implementation using lists.