S1.2: Higher Order Functions

CSci 2041:

Advanced Programming Principles

University of Minnesota,
Prof. Van Wyk,
Spring 2018

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:
   `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 -> 'a ->bool) -> 'a ->`
  `                      '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 function 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 S1.1 and how we can use a more general purpose `find_all_by` function instead.

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 `SamplePrograms` directory.

Can we write `find_all_by` ?

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

Revisiting the type of find

- Our intention, was that `find_all_by` had the type
  `('a -> 'a ->bool) -> 'a -> 'a list -> 'a list`
- What did OCaml infer as the type? It was
  `('a -> 'b -> bool) -> 'a -> 'b list -> 'b list`
  What does this mean? Why are there two type variables?
- OCaml is telling us that we can use this function in more general ways than we maybe expected.
- The elements of the list don't have to be the same type as the value we are looking for.
- We can the use this function in a variety of ways.
  See `find_and_lookup.ml`.

Other examples

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

- 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
  `fun` *formal parameters* `->` *body*
- *e.g.*
  - `fun x y -> x = y`
  - `fun x -> x + 1`
- Let’s define the equality function in `is_elem` using a lambda-expression.

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`
  - However, `::` is not treated this way. So `(::)` does not work.
- Let’s define the equality function in `is_elem` using a lambda-expression.

Use of curried functions

Recall the type of `find_all_by`

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

We could define a “default” find function as follows:

- `let find_all = find_all_by (=)`

What is the type of `find_all`?

Consider `find_all_with`

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

let equals x y =  x = y

let res_1 = find_all_with (equals 4) [1;3;5;4;6]

let res_2 = find_all_with ((=) 4) [1;3;5;4;6]
```

Note the use of curried functions in using `find_all_with`.

(These examples are all in `find_and_lookup.ml`.)

Comparing `find_all_by` and `find_all_with`

Which one of these should a library provide?

Which provides more opportunities for reuse?

We can create `find_all_by` using `find_all_with`:

- `find_all_by f v l = find_all_with (f v) l`

but not the other way around.

This suggests that `find_all_with` is more "reusable" in some sense and would be the one to include in a library if, for some reason, only one could be provided.

We can:

- `let find_all' v = find_all_with ((=) v)`

But, this is a bit more clumsy that our first `find_all`.

- `let find_all = find_all_by (=)`

We need to mention the argument `v` for `find_all'` but not for `find_all`.

So, neither `find_by` or `find_with` is obviously better than the other.

But we do want to understand the implications of the design of each one.

"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.)

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 list)`
- `min: ('a -> 'a -> int) -> 'a list -> 'a option`
- `max: ('a -> 'a -> int) -> 'a list -> 'a option`

Additional examples

Functions drop_while, drop_until:

- These have the type
  `'a list -> ('a -> bool) -> '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.

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
```

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
```

Some word games ...

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

We'll define out 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'; 'I
```

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

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 `estrings.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
           (string_to_estring "a b\t c d")
```

See examples in `filter.ml`

- Let's consider filters over strings and revisit `estrings.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.

Your solution should look something like the following:

```
let removeABCD cs =
  let notABCD = function
    | 'A' | 'B' | 'C' | 'D' -> false
    | _ -> true
  in
  filter notABCD cs
```

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
```

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
  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
  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 `fold.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?)

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.