Code Review 1 Solutions: February 7, 2020

Types and Higher Order Functions

Hakeem Angulu, hangulu@college.harvard.edu

Learning Goals

- Understand the goal of CS51
- Understand and use git effectively
- Define abstract and concrete syntax
- Understand OCaml's type system
- Understand and be able to implement the concept of recursion
- Describe functions as values, and understand the importance of that to good design
- Understand the types of errors

Why CS51?

CS51's official title is Computer Science 51: Abstraction and Design in Computation. Let's break that down:

- Computation: any type of calculation that includes both arithmetical and non-arithmetical steps and follows a well-defined model, for example an algorithm.
    - For example: thinking about the fastest route to get to Pierce, and a program that sums the elements of a list of integers
- Abstraction: the process of removing the parts of your algorithms that tend to inconsequential attributes of your data so as to allow for reusability
    - For example: redesigning a piece of code to accept lists of both integers and floats to allow for its use on both data types, and thus its reusability
- Design: the process of making your code beautiful and efficient. **There are many**

ways to solve a problem, but not all of them are equally good.

This class seeks to teach you how to write more robust, reusable, and beautiful code.

A Brief Introduction to Git

(adapted from https://thenewstack.io/tutorial-git-for-absolutely-everyone/)

Git is version-control software for working on group project collaboration. Not collaboration as in group dynamics and fair distribution of project workloads, but in the actual mechanics of sharing work back and forth. When a week's worth of shared project progress gets accidentally deleted or overwritten, or hours of work get eaten by a faulty file transfer, things can get messy.

Just as Google Drive makes it possible for multiple contributors to write, edit and add to the contents of a single text file, git is a computer program making it possible for multiple coders (and project managers, testers, content providers, and whomever else is on the team) to collaborate on a single project.

Learning git can be intimidating if you yourself have little or no programming experience, but just keep thinking of it as a funny-looking Google Docs and you'll do fine. Fun fact: git was invented by Linus Torvalds — the same Linus who created the Linux open-source operating system which now runs vast swaths of the internet, including Google and Facebook.

Git is essential to your programming workflow, so please take some time to learn how to use it (well). You'll use it in future classes, in research, and in industry.

An excellent, no-frills guide that assumes no git experience is located here, and another is located started on page 213 of our textbook, *Abstraction in Design and Computation*.

Here is a quick list of the git commands most relevant to you right now:

- To clone (download) a remote (hosted on the web on a platform like GitHub or BitBucket) repository (a directory of files for a project), run in your terminal:

```
git clone [remote repository url] [folder name]
```

This will download the repository onto your local machine, into a folder called the [folder name]. If you don't specify a folder name, it will create a folder called the name of your remote repository.

- To track (have git monitor for changes) all the files in your repo, run in terminal at the cloned repository:

  ```
  git add -A
  ```

- To save a copy of the tracked work, run:

  ```
  git commit -am "[message]"
  ```

  Where the [message] is some sort of short descriptive text that outlines the changes you've made. Examples include "started pset0" and "fixed problem 2".

- To push (upload) a commit (the saved copy of the tracked work) to the remote repository you cloned from, run:

  ```
  git push
  ```

- To download updates from the remote repository, run:

  ```
  git pull
  ```

Git is a very powerful piece of software that you'll get more comfortable using over the course of CS51.

Types and Type Inference

OCaml is a strong, statically typed language. Let's break that down:

- Strong typing means that each type can only be used in certain ways becuase every function can only input and output certain types.
- Static typing means that each expression’s type can be evaluated by looking at the expression in context. This is compared with *Dynamic typing* whereby the types are figured out as the program runs.

Strong and static typing allows OCaml to detect type errors at compile time. OCaml is also *implicitly typed*. Via a process called *type inference*, the compiler can look at a program and figure out what the type of each variable is. However, type inference isn't perfect, and is an area of ongoing research, and you should be in the practice of adding type annotations to your code to help OCaml.

Think about the following exercises from Lab 1:

```
let exercise6c : ??? =
  fun x -> x +. 11.1 ;;
```

```
let exercise6b : ??? =
  let greet y = "Hello " ^ y
  in greet "World!";;
```

Exercise 1: What is the type of exercise6c?

```
float -> float
```

Exercise 2: What is the type of exercise6b?

```
string
```

Now, let's look at a slightly more complicated function:

```
let rec f1 (x : ???) (y : ???) : ??? =
  match y with
  | [] -> []
  | _hd :: tl -> (x, y) :: (f1 x tl) ;;
```

Exercise 3: Rewrite this function, filling in the types denoted by "???".

```
let rec f1 (x : 'a) (y : 'b list) : ('a * 'b list) list =
  match y with
  | [] -> []
  | _hd :: tl -> (x, y) :: (f1 x tl) ;;
```

Recursion

By now, you're all familiar with the concept of recursion, simply defined as: when a thing (in our case, a function) is defined in terms of itself. An example is the function from Exercise 3 above. Recursive functions are easy to spot in OCaml because of the necessity of the `rec` syntax. Recursion can be confusing, especially to people with background in iterative/procedural programming (like that taught in the bulk of CS50), but you'll become more comfortable with this concept with practice.

A popular example of recursion is the calculation of an integer's factorial:

```
let rec factorial (x : int) =
  if x = 0 then 1
  else x * factorial (x - 1) ;;
```

Can you see why this is recursive and why this works?

Higher Order Functions

OCaml treats functions themselves as **values**, and thus, they can be passed to other functions as arguments. This allows for the creation of higher-order functions, which reduce repeated code.

Higher order functions take functions as their arguments in order to perform some computation. The higher order functions you've been introduced to are:

(adapted from Albert Zheng)

- Map

  ```
  let rec map f lst =
  match lst with
  | [] -> []
  | hd :: tl -> (f hd) :: (map f tl) ;;
  ```

  When to use: Map takes a list and returns a list, so if your goal is convert a list into a new list of the same length, Map may be helpful.

- Fold left

  ```
  let rec fold_left f acc lst =
  ```

```
  match lst with
  | [] -> acc
  | hd :: tl -> fold_left f (f acc hd) tl ;;
```

When to use: Fold left takes a list and returns a single value, so if your goal is to return a single value based on the elements of a list, Fold left may be helpful.

- Fold right

```
 let rec fold_right f lst acc =
  match lst with
  | [] -> acc
  | hd :: tl -> f hd (fold_right f tl acc) ;;
```

Fold right is essentially the same as Fold left, but Fold right goes through the list from right to left while Fold left goes from left to right. When to use: Only use Fold right if you must go through a list from right to left (otherwise use Fold left)

- Filter

```
 let rec filter f lst =
  match lst with
  | [] -> []
  | hd :: tl -> if f hd
                then hd :: filter f tl
                else filter f tl ;;
```

When to use: Filter takes a list and returns a subset of that list, so if your goal is to take a list and return a new list that only contains some of the elements from the original, then Filter may be useful.

You will not have to implement these functions every time. To call them, simply write (filling in the arguments):

```
List.Filter _ _
```

I highly suggest getting very well acquainted with (and bookmarking) the following two OCaml documentation pages:

- Pervasives module: A list of all the functions that come built into OCaml

- List module: A list of all the functions in the list module (including all the higher order functions above)

Those will be your best friends when figuring out which functions take which inputs and what they give as outputs. In addition, being able to read and understand the documentation of the language you're using is a crucial skill to have as a programmer.

Finally, let's do two more exercises (adapted from Albert Zheng):

Exercise 4: Using a higher-order function, define a function ”special_sum” that takes an int list ”lst” and returns the sum of the elements in ”lst” that are greater than 5. For example:

```
> special_sum [4; 5; 6; 7] ;;
>>> 13
```

```
let special_sum (lst : int list)=
  List.fold_left (fun acc x -> acc + x) 0 (List.filter (fun x -> x > 5) l
```

Exercise 4: Using a higher-order function, define a function ”is_mult_3” that takes an int list ”lst” and returns a new int list whose elements are the elements of ”lst” that are multiples of 3. For example:

```
> is_mult_3 [0; 2; 3; 5; 6; 12] ;;
>>> [0; 3; 6; 12]
```

```
let is_mult_3 (lst : int list)=
  List.filter (fun x -> x mod 3 == 0) lst ;;
```

For this and future code reviews, please do not hesitate to reach out to me with any questions or concerns. My email can be found at the top of this document.