Code Review 5: March 6, 2020

Midterm Review

Hakeem Angulu, hangulu@college.harvard.edu

Tips

- Do the labs.
    - Carefully read through the comments on the solutions. They provide a lot of insight.
- Look back through the Code Review notes (both mine and everyone else's).
- Design and style will matter on the midterm.
- Practice writing code by hand.
- Answer questions in the order that makes most sense to you -- not necessarily sequentially. If you're stuck on a question, move on and do it later.
- Everything from the first half of the class (up to Lab 10) is fair game for the exam.
- Redo the labs.

Types and Type Inference

Exercise 1: Give the type inference for the following functions. If it does not type, explain why.

a)

```
let f1 x y z =
  match x with
  | [] -> y
  | _ :: _ -> List.fold_left z y x ;;
```

```
f1: _
```

b)

```
let f2 x y = f2 3 4. ;;
```

```
f2: _
```

c)

```
let f3 x y =
  [1 + x;  y] ;;
```

```
f3: _
```

d)

```
let f4 x y =
  match (g x) with
  | [] -> 2
  | _ :: tl -> (g tl) ;;
```

```
f4: _
```

e)

```
let f5 x =
  match x with
  | None -> 0
  | Some y -> let f = fun z -> z + 1 in
              f y
  in
    f (Some 41) ;;
```

```
f5: _
```

Exercise 2: Write a function with no explicit typing such that the OCaml compiler will infer the given type.

a)

```
f1: int -> 'a -> float -> 'a option
```

```
let f1 =
 failwith "f1 not implemented" ;;
```

b)

```
f2: string -> bool -> bool option
```

```
let f2 =
 failwith "f2 not implemented" ;;
```

c)

```
f3: ('a -> bool) -> 'a list -> 'a list
```

```
let f3 =
 failwith "f3 not implemented" ;;
```

d)

```
f4: int option -> (int -> int) -> int option
```

```
let f4 =
 failwith "f4 not implemented" ;;
```

Variant Types

Exercise 3: Define a variant type for a Computer Science class at Harvard.

```
type cs = _
```

Exercise 4: Define a type for a student at Harvard who only takes 4 CS classes. Important information include their name, class year, and CS classes.

```
type student = _ ;;
```

Higher Order Functions

Exercise 5: The Bureau of Study Counsel found that students tend to come for help in threes, fours, or alone. Define a type, `study_group` that captures this information.

```
type study_group = _
```

Exercise 6: The BSC has written some function to address the students' concerns with their CS classes. Write a function to map that function over a given study group. Do not use List.map.

```
let group_map (f : student -> student) (g : study_group) : study_group =
  failwith "group_map not implemented" ;;
```

Exercise 7: Now, the BSC has decided to count the number of occurrences of CS51 in students' schedules. Write a function to do this without functions in the List module. Students may enroll in the same class several times.

```
let cs51_trouble (g : study_group) : int =
  failwith "cs51_trouble not implemented" ;;
```

Modules and Functors

Exercise 8: (adapted from Albert Zheng) The world's leading discombobulation scientist Dr. Shuart Stieber has

made an incredible breakthrough! He has figured out how to convert any cow into a dragon (or any dragon into a cow) using his newly patented invention: the Recombobulator. Now, Dr. Stieber wants to use his new device on a cow to convert it into a dragon, then use his device on the dragon to convert it back in to a cow (and so on). He plans to keep track of whether the cow is currently a cow or a dragon using modules (I mean, how else would you keep track of it?). Below we have defined a module signature for the module CowOrDragon which will be our initial representation of the cow that Dr. Stieber will be Recombobulating.

```
module type CowOrDragonSig =
  sig
    type animal
    val numWings : int
    val hasLongNeck : bool
    val hasScales : bool
    val startedOutAs : animal
    val currentForm : animal
    val convert : animal -> animal
  end


module CowOrDragon : CowOrDragonSig =
  struct
    type animal = | Cow | Dragon
    let numWings = 0
    let hasLongNeck = false
    let hasScales = false
    let startedOutAs = Cow
    let currentForm = Cow
    let convert x = if x = Cow then Dragon else Cow
  end
```

Define a functor, `Recombobulate`, which takes in a module of type `CowOrDragonSig` and returns a new module of the same type, but: if the original module had the attributes of a Cow, the new module must have the attributes of a Dragon.

```
module Recombobulate (C : CowOrDragonSig) : CowOrDragonSig =
  struct

    _
  end
```

Students and CS, revisited (adapted from Ezra Zigmond)

Suppose you've been hired by SEAS to implement a student directory application. You know that you'll need a way to represent students, and a data structure that you can use to store information about students and look them up.

Exercise 9: Suppose that the only majors in SEAS are CS, EE, and ME. Define a type major to represent this.

```
type major = _ ;;
```

Exercise 10: Suppose that a student is defined by an integer id and a major. Define a type student to represent a student.

```
type student = _
```

Conveniently, your friend tells you that they have implemented a dictionary data structure! (Recall that a dictionary, also called a hash table or a map, lets you look up values according to some key.) Your friend tells you that the dictionary is incredibly efficient – it uses machine learning or block chain or something, you think. In any case, your friend hasn't shown you the implementation and quite honestly, you're not sure you want to see it. You do, however, know the following module signatures.

```
module type Dict =
  sig
    type dict
    type key
    type v
    val empty : dict
    val add : key -> v -> dict -> dict
    val fold : (key -> v -> 'a -> 'a) -> 'a -> dict -> 'a
  end

module type DictArg
  sig
    type key
    type v
  end
```

Your friend has also defined a functor for creating dictionaries that looks like this:

```
module MakeDict (DA : DictArg) : (Dict with type key = DA.key and type v = DA.v) =
  struct
    (* Implementation goes here... *)
  end
```

Exercise 11: Define a module StudentDictArg with signature DictArg that would be suitable to pass to the MakeDict functor to create a dictionary that maps student names (as strings) to students.

```
module StudentDictArg = _
```

Exercise 12: Use the `MakeDict` functor to construct a module `StudentDir`.

```
module StudentDir = _
```

Exercise 13: The `fold` function provided by the Dict signature can be used analogously to the `List.fold_left` function to fold a function over all of the keys and values in a dictionary. Suppose that we want to implement a function `count_cs` that takes in a student directory dictionary and returns the number of CS majors in the dictionary using the fold function. Use fold to implement `count_cs`.

```
let count_cs : StudentDir.dict -> int =
  failwith "count_cs not implemented" ;;
```

Exercise 14: Write an implementation of the `MakeDict` functor. It can be simple.

```
module MakeDict (DA : DictArg) : (Dict with type key = DA.key and type v = DA.v) =
  struct

    _
  end
```

The solutions to all of the above exercises will be available on Saturday at noon to give you some time to work through the solutions in preparation for the midterm.

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.