Code Review 5 Solutions: 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 8) 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: a' list -> 'b -> ('b -> 'a -> 'b) -> 'b
```

b)

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

```
(* This does not typecheck. f2 needs to have a rec flag
   to be called within itself. *)
```

c)

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

```
f3: int -> int -> int list
```

d)

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

```
(* Write the type below *)
(* This function is not well typed. The first case returns an int and
   the second case returns a list. *)
```

e)

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

```
(* We observe that x must be an option type, since it matches with an
   option type. We add 1 to its result, so it must be an int option.
   Finally, it returns an int, observed from the first match case. *)
f5:  int option -> int
```

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 i arg f =
  if float_of_int(i) > f then Some arg else None ;;
```

b)

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

```
let f2 a b =
  if a = "hello" && b then Some b else None ;;
```

c)

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

```
f3 = List.filter ;;
```

d)

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

```
let f4 a b =
  match a with
  | None -> None
  | Some v -> Some (y v) ;;
```

Variant Types

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

```
type cs =
| CS1
| CS20
| CS50
| CS51
| CS61
| CS121
| CS124
| CS181
| CS182
;;
```

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 =
  {name : string; year : int; classes = cs list} ;;
```

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 =
| Solo of student
| Triplet of student * student * student
| Quad of student * student * student * student ;;
```

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 rec group_map (f : student -> 'student') (g : study_group) : study_group =
  match g with:
  | Solo s -> f s
  | Triplet (a, b, c) -> Triplet (f a, f b, f c)
  | Quad (a, b, c, d) -> Quad (f a, f b, f c, f d) ;;
```

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 not enroll in the same class several times.

```
let cs51_trouble (g : study_group) : int =
  let rec count_cs51 (s : student) : int =
    match s.classes
    | [] -> 0
    | hd :: tl -> if hd = CS51 then 1 else count_cs51 tl in
  match g with
  | Solo x -> count_cs51 x
  | Triplet (x, y, z) -> count_cs51 x + count_cs51 y + count_cs51 z
  | Quad (w, x, y, z) -> count_cs51 w + count_cs51 x + count_cs51 y
                         + count_cs51 z
```

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
    type animal = C.animal
    let numWings = 2 - C.numWings
    let hasLongNeck = not C.hasLongNeck
    let hasScales = not C.hasLongNeck
    let startedOutAs = C.startedOutAs
    let currentForm = C.convert C.currentForm
    let convert = C.convert
  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 = CS | EE | ME ;;
```

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 = {id : int; major : major}
```

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 : (DictArg with type key = string and type v = student) =
  struct
    type key = string
    type v = student
  end
```

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

```
module StudentDir = MakeDict(StudentDictArg)
```

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 =
  StudentDir.fold (fun _ v acc -> if v.major = CS then acc + 1 else acc) 0 ;;
```

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
    type key = DA.key
    type v = DA.v
    type dict = (key * v) list
    let empty = []
    let add k v d = (k, v) :: d
    let fold f init d = List.fold_left (fun acc (k, v) -> f k v acc) init d
  end
```

Notes on Substitution Semantics

Free Variables

Variable occurrences can be said to be either free or bound. A bound occurrence of a variable falls within a construct (like a let expression or a function) that introduces that variable. Otherwise, the variable is free. In an expression like `x + 8`, for example, `x` is free; in an expression like `let x = 2 in x + 8`, `x` is bound by the let expression.

```
(* The set of free variables for an integer expression m is the empty set. *)
FV(m) = ∅
```

- FV(3) = ∅
- FV(-2) = ∅

```
(* The set of free variables for a variable is the set containing that variable. *)
FV(x) = {x}
```

- FV(x) = {x}
- FV(y) = {y}

```
(* The set of free variables for an addition expression P + Q is the union of the free v
FV(P + Q) = FV(P) ∪ FV(Q)
```

- FV(x + 3) = {x}
- FV(x + y) = {x, y}

```
(* The set of free variables for an function application P Q is the union of the free va
FV(P Q) = FV(P) ∪ FV(Q)
```

- FV(f 2) = {f}
- FV(f x) = {f, x}

```
(* The set of free variables for a function fun x -> P is the set of free variables of P
(* This is because the function fun x -> P binds the variable x inside of P. *)
FV(fun x -> P) = FV(P) - {x}
```

- FV(fun x -> x + 10) = ∅
- FV(fun x -> x + y) = {y}

```
(* The set of free variables for a let expression `let x = P in Q` is the set of free va
(* This is because the let expression binds the variable x inside of Q. *)
FV(let x = P in Q) = (FV(Q) - {x}) ∪ FV(P)
```

- FV(let x = 3 in x + 1) = ∅
- FV(let x = 3 in x + y) = {y}
- FV(let x = x + 2 in x + y) = {x, y}

Substitution

The expression `E[x -> P]` substitutes free occurrences of `x` in the expression `E` with the expression `P`.

```
(* Substituting `x` for `P` in an integer expression leaves the expression unchanged. *)
m[x -> P] = m (where m is an integer)
```

- 2[x -> 3] = 2
- 5[x -> 5] = 5

```
(* Substituting `x` for `P` in a variable expression `x` results in `P`. *)
x[x -> P] = P
```

- x[x -> 2] = 2
- x[x -> 8] = 8

```
(* Substituting `x` for `P` in a variable expression `y` (a different variable) results
y[x -> P] = y
```

- y[x -> 2] = y
- y[x -> 8] = y

```
(* Substituting `x` for `P` in an addition expression `Q + R` requires substituting in
(Q + R)[x -> P] = Q[x -> P] + R[x -> P]
```

- (x + 3)[x -> 2] = 2 + 3
- (x + y)[x -> 2] = 2 + y

```
(* Substituting `x` for `P` in a function expression `fun x -> Q` leaves the function e
(fun x -> Q)[x -> P] = fun x -> Q
```

- (fun x -> x)[x -> 2] = (fun x -> x)
- (fun x -> x + 5)[x -> 10] = (fun x -> x + 5)

```
(* Substituting `x` for `P` in a function expression `fun y -> Q`
 where `y` is not a free variable of `P` results in a substitution in `Q`.*)
(fun y -> Q)[x -> P] = fun y -> Q[x -> P]
(* where `x` and `y` are different variables and `y` is not in `FV(P)` *)
```

- (fun y -> x + y)[x -> 2] = (fun y -> 2 + y)
- (fun y -> x)[x -> 8] = (fun y -> 8)

```
(* Substituting `x` for `P` in a function expression `fun y -> Q` where `y` is a free v
  requires a substitution of `y` for a fresh variable `z` before a substitution in `Q`.*
(fun y -> Q)[x -> P] = fun z -> Q[y -> z][x -> P]
(* where `x` and `y` are different variables, `y` is in `FV(P)`, and `z` is a fresh vari
```

- (fun y -> 3)[x -> y] = (fun z -> 3)
- (fun y -> x)[x -> y] = (fun z -> y)
- (fun y -> y)[x -> y] = (fun z -> z)
- (fun y -> x + y)[x -> y] = (fun z -> y + z)

```
(* Substituting `x` for `P` in a let expression `let x = Q in R` results in a substitut
(let x = Q in R)[x -> P] = let x = Q[x -> P] in R
```

- (let x = 2 in 3)[x→3] = let x = 2 in 3
- (let x = x + 1 in 3)[x→3] = let x = 3 + 1 in 3
- (let x = x + 1 in x)[x→3] = let x = 3 + 1 in x
- (let x = x + 1 in x)[x→y] = let x = y + 1 in x

```
(* Substituting `x` for `P` in a let expression `let y = Q in R` where `y`
 is not a free variable of `P` results in a substitution in both `Q` and `R`.*)
(let y = Q in R)[x→P] = let y = Q[x→P] in R[x→P]
(* where `x` and `y` are different variables and `y` is not in `FV(P)` *)
```

- (let y = 2 in 3)[x→3] = let y = 2 in 3
- (let y = x + 1 in y)[x→3] = let y = 3 + 1 in y
- (let y = x + 1 in x + y)[x→3] = let y = 3 + 1 in 3 + y
- (let y = x + 1 in x + y)[x→z] = let y = z + 1 in z + y

```
(* Substituting `x` for `P` in a let expression `let y = Q in R` where `y` is a free va
 in a substitution in both `Q` and `R`, but `R` first must have a substitution of `y` fo
(let y = Q in R)[x→P] = let z = Q[x→P] in R[y→z][x→P]
(* where `x` and `y` are different variables, `y` is in `FV(P)`, and `z` is a fresh vari
```

- (let y = 2 in 3)[x→y] = let z = 2 in 3
- (let y = 2 in y)[x→y] = let z = 2 in z
- (let y = x + 1 in y)[x→y] = let z = y + 1 in z
- (let y = x + 1 in x + y)[x→y] = let z = y + 1 in y + z

Evaluation

```
(* The evaluation of a number n is just that number n. *)
n ⇓ n
```

- 5 ⇓ 5
- 1 ⇓ 1

```
(* The evaluation of `P + Q` involves evaluating P to a value, evaluating Q to a value,
P + Q ⇓
      | P ⇓ m
      | Q ⇓ n
      ⇓ m + n
```

- 2 + 8 ⇓ 10
- 5 + 1 ⇓ 6

```
(* The evaluation of `let x = D in B` involves evaluating D to a value v_D.
   Then, we actually need to perform the substitution, replacing all free occurrences of
   with `that value `v_D` until we get a new value `v_B`.
*)
let x = D in B ⇓
               | D ⇓ v_D
               | B[x -> v_D] ⇓ v_B
               ⇓ v_B
```

```
let x = 2 + 8 in x + 5 ⇓
                       | 2 + 8 ⇓
                               | 2 ⇓ 2
                               | 8 ⇓ 8
                               ⇓ 10
                       | 10 + 5 ⇓
                                | 10 ⇓ 10
                                | 5 ⇓ 5
                                ⇓ 15
                       ⇓ 15
```

```
(* The evaluation of a function `fun x -> B` is just the function itself. *)
fun x -> B ⇓ fun x -> B
```

- (fun x -> 10) ⇓ (fun x -> 10)

```
(* The evaluation of a function application `P Q` involves evaluating `P` to a function
   From there, the argument of the function `Q` must be evaluated to a value `v_Q`.
   Then, we take the body of the function `B`, and substitute `x` with that value `v_Q`,
*)

P Q ⇓
    | P ⇓ fun x -> B
    | Q ⇓ v_Q
    | B[x -> v_Q] ⇓ v_B
    ⇓ v_B
```

```
(fun x -> x + 4) (1 + 2) ⇓
                         | (fun x -> x + 3) ⇓ (fun x -> x + 3)
                         | 1 + 2 ⇓
                                 | 1 ⇓ 1
                                 | 2 ⇓ 2
                                 ⇓ 3
                         | 3 + 4 ⇓
                                 | 3 ⇓ 3
                                 | 4 ⇓ 4
                                 ⇓ 7
                         ⇓ 7
```

Substitution

Exercise 15: (adapted from Katherine Binney) What is the type of the following expression, and what does it evaluate to?

```
let f g h = fun x -> h (g x) in
let f = f (fun x -> (x, x)) in
let f = f (fun x -> let (x, _) = x in x + x) in
f 3 ;;
```

The solution is provided below:

First, replace all the variables with unique ones to make the expression easier to understand:

```
let f1 g1 h1 = fun x1 -> h1 (g1 x1) in
let f2 = f1 (fun x2 -> (x2, x2)) in
let f3 = f2 (fun x3 -> let (x4, _) = x3 in x4 + x4) in
f3 3 ;;
```

Now, substitute definitions:

```
f3 3
  [f2 (fun x3 -> let (x4, _) = x3 in x4 + x4)] 3
  [[f1 (fun x2 -> (x2, x2))] (fun x3 -> let (x4, _) = x3 in x4 + x4)] 3
  [fun g1 -> fun h1 -> fun x1 -> h1 (g1 x1)] (fun x2 -> (x2, x2))
            (fun x3 -> let (x4, _) = x3 in x4 + x4) 3

(*
  We changed f1 away from syntactic sugar,
  Now, we can start to apply our functions: Note we have a
  function that takes 3 args, and we have 3 args! (two are functions)

  g1 is (fun x2 -> (x2, x2))

  h1 is (fun x2 -> (x2, x2)) (fun x3 -> let (x4, _) = x3 in x4 + x4)

  x1 is 3
*)

[(fun x3 -> let(x4, _) = x3 in x4 + x4)] ([(fun x2 -> (x2, x2))] 3)

(*
  Now we have 2 nested functions, the inner one being applied to 3 - which means x2 = 3)
  (fun x3 -> let (x4, _) = x3 in x4 + x4) (3, 3)
  let (x4, _) = (3, 3) in x4 + x4
  6
```

Hence, the type of this expression is `int`, and its value is `6`.

Looking at each expression within the larger expression on its own:

```
f1 : ('a -> 'b) -> ('b -> 'c) -> 'a -> 'c
(*
  f1 takes in two functions and applies them to a 3rd argument in
  swapped order. So if x is of type 'a, g will make it into 'b, then
  h must take in that thing and can return a 3rd type
*)

f2: ('a * 'a -> 'b) -> 'a -> 'b
(*
  f2 partially applies f1 to the function fun x2 -> (x2, x2). This anonymous
  function has type 'a -> 'a * 'a, which means the previously polymorphic 'b
  is now actually of type 'a * 'a. Because of partial application,
  we "have" the first arg to f1, and return the function from 2nd arg
  to output
*)

f3 : int -> int
(*
  f3 partially applies f2 to the function fun x3 -> let (x4, _) = x3 in
  x4 + x4. This function must be of type ('a * 'a -> 'b), because it is
  the first arg to f2. If we look at the implementation, we see we add
  the first element in the tuple using +, meaning 'a must be an int,
  and the function returns an int.
*)
```

Note: this problem is significantly more challenging than I would expect you to see usually, but I think going through it is a good exercise in understanding types and substitution.

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.