Code Review 5 Solutions: Midterm Review

CS51
Melissa Kwan

1 Midterm strategies

- To review, redo labs on your own! The exam will be much more like the labs than the problem sets.
- Have utop and OCaml open as you take the midterm. Write your functions in the OCaml file, then copy and paste them into Gradescope. I recommend prepping your OCaml file with some reference syntax, such as an example module/functor and recursive function that uses match statements.

2 Recursion and higher-order functions

Question 2.1. Write a function `map_drop` that takes in three inputs: a list, a function, and an element. Thus function will behave as a standard map with one caveat — if the element is in the list, it will drop that element from the list.

```
(* Solution *)
let map_drop f el lst =
    List.map f (List.filter (( = ) el) lst)
```

Question 2.2. Write a function `replicate` that takes in a list of strings and an integer and returns a new list with the elements replicated a given number of times. For example `replicate ["a";"b";"c"] 3` would yield `["a"; "a"; "a"; "b"; "b"; "b"; "c"; "c"; "c"]`.

```
(* Solution *)
let replicate lst n =
    let rec append n acc x =
        if n = 0 then acc
        else append (n-1) (acc @ [x]) x in
    List.fold_left (append n) [] lst ;;
```

3 Modules, Functors, Sharing Constraints

TL; DR on Modules / Functors / Signatures:

- A **module** is a package of types and values. (Remember that functions are values, too.)
- A **signature** is a type of a module. It defines what the outside user can see, and it helps us set up an *abstraction barrier* between the "interface" (i.e. how the user interacts with your module) and the underlying implementation.
- A **functor** is a function from modules to modules.
- A **sharing constraint** allows us to reveal information that is not accessible from the signature.

3.1 The Recombobulator (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
```

Question 3.1. 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, and vice versa.

Solution: Note that we do *not* need a sharing constraint here. `Recombobulate` does not actually need to know that the type of an `animal` is a `Cow | Dragon`: it can just piggyback off of all of the input signature's original functions. And it's revealed in the signature that `numWings` is an integer and `hasLongNeck` and `hasScales` are booleans, so we have no problem using integer and boolean operations for those values.

```
(* SOLUTION *)
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
```

3.2 Students and majors

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.

Question 3.2. Suppose that the only majors in SEAS are CS, EE, and ME. Define a type major to represent this.

```
(* SOLUTION *)
type major = CS | EE | ME
```

Question 3.3. Suppose that a student is defined by an string name and a major of type major. Define a type student to represent a student.

```
(* SOLUTION *)
type student = {name : string; 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 input keys.) 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 lookup : key -> dict -> v option
    end

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

Question 3.4. Define a functor called `MakeDict` that takes in a `DictArg` and and outputs a dictionary. Under the hood, implement your dictionary as a list of tuples (which is probably what your friend is doing under all that smoke and mirrors).

Solution: Why do we need a sharing constraint? For the output of `MakeDict` on a `DictArg` to be useful at all, the user needs to be able to add keys and values into the resulting dictionary.

Imagine you're the user, and you created this special module of type `DictArg` with `type key = string` and `type v = int`. All you want to do is to make a dictionary with string keys and integer values. You call the functor to create the dictionary. Everything seems fine. But then you try to add a pair of a string and an integer to your dictionary, and OCaml tells you that you're not allowed to know the type of `key` and `v`. You would be understandably peeved.

To address this issue, just add a sharing constraint. Now OCaml knows that it's allowed to disclose the "implementation" of `key` and `v` to the outside world. Problem solved.

```
(* SOLUTION *)
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 (pair : key * v) (d : dict) : dict = pair :: d
        let rec lookup (target_k : key) (d : dict) : v option =
            match d with
```

```
            | [] -> None
            | (k, v) :: tl -> if k = target_k then Some v
                               else lookup target_k tl
    end
```

Exercise 3.5. We want to create a dictionary that maps student IDs (as ints) to students. Define an input module `StudentDictArg` with signature `DictArg` that we could pass to the `MakeDict` functor to accomplish this.

```
(* SOLUTION *)
(* Defining the input module *)
module StudentDictArg : (DictArg with type key = int and type v = student) =
    struct
        type key = int
        type v = student
    end

(* ALTERNATE SOLUTION has no signature, and needs no sharing constraints *)
module StudentDictArg =
    struct
        type key = int
        type v = student
    end
```

Exercise 3.6. Use the `MakeDict` functor to create a module called `StudentDir` that maps student IDs (as ints) to students.

```
(* SOLUTION *)
(* Calling the functor on the module *)
module StudentDir = MakeDict(StudentDictArg)
```

Exercise 3.7. Create a dictionary called `me` that maps your HUID to your student record. (Apologies if your major isn't covered.)

```
(* SOLUTION *)
open StudentDir ;;
let me = add (123456, {name="Melissa Kwan"; major=CS}) empty ;;
```

4 Data Structures Review

Recall again our binary tree data structure:

```
type 'a bintree =
    | Leaf
    | Node of 'a * 'a bintree * 'a bintree ;;

let rec leaf_count (tree : 'a bintree) : int =
    match tree with
    | Leaf -> 1
    | Node (_, left, right) -> leaf_count left + leaf_count right ;;
```

Exercise 4.1. Write a function `flatten_tree` that takes in a binary tree and returns a list of the same elements in the binary tree. The order of elements in the list is the Depth-First-Search order of the tree (i.e. go as far down as you can before exploring horizontally.)

```
let rec flatten_tree (root : 'a bintree) : 'a list =
    match root with
    | Leaf -> []
    | Node (el, leftTree, rightTree) ->
        flatten_tree left_tree @ flatten_tree right_tree @ el ;;
```

5 Supplementary Type Questions

These types of questions will probably not be on the 2022 midterm, since the online + open utop format trivializes type inference questions. However, they can be useful to reason about by hand!

5.1 Type inference

For these expressions, give the most general type, as would be inferred by OCaml. If an expression does not typecheck, explain why. First, look at how many arguments there are, and then see what each line implies about the arguments. Here are some questions to ask yourself:

- **"Which operators are being used?"** Operators like `::` or `+` or `||` can give you hints about the type of the function. Conversely, if you're given vague operators like `=`, you can only infer polymorphic types.
- **"Are all output types the same? Is the function trying to use one input with different type operators?"** These are sanity checks that can help you determine whether it will type check.

1.
```
let f1 x =
    match x with
    | None -> raise (Invalid_argument "beep")
    | Some y ->
```

```
        match y with
        | [] -> raise (Invalid_argument "boop")
        | h :: t -> Some (h []) ;;

(* SOLUTION: ('a list -> 'b) list option -> 'b option *)
```

2.
```
    let f2 x y =
        match x with
        | 1 -> y +. 2.
        | _ -> x + 3 ;;

(* SOLUTION: Does not type check because one match case returns a float,
while the other returns an int. *)
```

5.2 Type creation

Define a function using no explicit type annotation, which means no use of the `:` operator, such that OCaml will infer this type. The functions you write do not need to be practical, they just need to have the given type.

How to frame the problem:

1. These problems will most often be functions. Note the number inputs and outputs. Check if the output itself is a function.
2. Sometimes, you will be given specific types, like `int` or `bool option list` or `float * string -> string list`. The trick is to use an operator that only applies to that one type. If you want OCaml to infer something as a float, use the `+.` operator. For an option, match with `Some` or `None`. You get the idea.
3. In the trickier case, you'll be asked to create a function of polymorphic types. That means that you have to avoid any operators that will "give away" the type. Let's look at the example `'a list -> 'a -> 'b`. Here are the questions to ask yourself.
    - **"Which inputs need to be of the same type?"** If you're given `'a list -> 'a -> 'b`, you need to know that the two inputs have the same element type. You can show OCaml that two elements are of the same type without giving away the type by using the `=` operator.
    - **"Which types shouldn't be inferred as the same?"** If you're given `'a list -> 'a -> 'b`, you can't just output something that clearly has the type of the input. If you need to pull a new type out of nowhere, it's a safe bet to raise an error.
    - "**Which types are semi-specified?"** You might need to generate something of type `'a list` or `'a option`. Your best bet here is to use the emptiest version of those structures: (`[]` in the list case and `None` in the option case). You want OCaml to infer the construct without inferring the type.
    - Here's our final result!

```
let f a1 a2 =
    match a1 with
    | [] -> raise (Invalid_argument "empty")
    | hd :: tl -> if hd = a2 then raise (Invalid_argument "same")
                  else raise (Invalid_argument "different") ;;
```

Try the problems below:

```
1. int -> bool -> bool
```

```
(* SOLUTION *)
let f1 x y = y && x < 0 ;;
```

```
2. string -> bool * bool -> bool list
```

```
(* SOLUTION *)
let f2 x (y, z) = if y && z then [] else [bool_of_string x; y; z] ;;
```

```
3. int option -> (int -> int) -> int option
```

```
(* SOLUTION *)
let f3 x y =
  match x with
  | None -> None
  | Some v -> Some (y (v + 1)) ;;
```

```
4. ('a -> 'b) -> 'a -> 'b
```

```
(* SOLUTION *)
let f4 x y = x y ;;
```

```
5. 'a -> bool
```

```
(* SOLUTION *)
let f5 x = true ;;
```

```
6. ('a * 'b -> 'c) -> 'a -> 'b -> 'c
```

```
(* SOLUTION *)
let curry uncurried x y = uncurried (x, y) ;;
```