Code Review 3 Solutions: February 21, 2020

Algebraic Data Types, Variants, and Invariants

Hakeem Angulu, hangulu@college.harvard.edu

Learning Goals

- Understand when to use helper functions
- Describe the kinds of data types in OCaml and their differences
- Describe the algebraic data type
- Construct and deconstruct algebraic data types with functions
- Enforce invariants

Logistical Notes

Problem Set 1 resubmissions are available - your design and style score may only increase if your correctness score does not decrease - addressing all the feedback will not guarantee an increased grade

---

A Brief Discussion of Helper Functions

There was a lot of feedback from the last problem set that the rules around the use of helper functions was unclear. To address that feedback, here are a few notes, some of which come directly from Professor Shieber:

- Avoid the term "helper function"
  - "All functions should be helpful. If it's not helpful, it probably isn't or shouldn't be a function." – Professor Shieber
  - The takeaway here is that helper functions can be good, and their use is sometimes encouraged. In lieu of "helper functions," I encourage you to use the term "auxiliary function" to encompass the same point
- The CS51 staff is not against the use of auxiliary functions - a huge amount of functionality would be significantly less clear or impossible without them
  - This is likely the most important point. We want to discourage the misuse of auxiliary functions.
  - The misuse comes when the auxiliary functions replace more idiomatic OCaml, like simple pattern matching and data structure deconstruction

Here are some times when we encourage the use of auxiliary functions:

1. Refactoring: Add auxiliary functions if doing so eliminates nontrivial redundancies. This refactoring is one of the main benefits of functions, and is a key principle of abstraction and clean code
   - If you have to write a very strange or complex pattern match that may be better implemented with an auxiliary function, use your judgment and implement the auxiliary function
2. Clarity: Even a helper function that is called only once can be useful, if splitting up the combination in

that way adds clarity to the code (by separating out the parts of a computation or by allowing for function and variable names that help document intent). But a single-use helper function that doesn't add substantial clarity may be better off eliminated.
3. Scope: Localize specialized helper functions that aren't of more general utility to where they are needed, instead of, for instance, defining them globally.
    - This was a huge problem in the problem set. In general, if you are calling a function once for another function's implementation, you should define the first function within the second to limit the scope of that function
    - This hides your function from other aspects of the program, which becomes useful, and critically important, in larger and more secure projects

As usual, if you're unsure, post a question on Piazza or reach out to me.

Now, here is an exercise for a function that likely cannot be written well without a helper function: the calculation of variance.

Exercise 1: Define a function `variance : float list -> float option` that returns `None` if the list has fewer than two elements. Otherwise, it should return the variance of the numbers in its list argument.

```
let variance (lst : float list) : float option =
  let sum, length = List.fold_left (+.) 0. lst, List.length lst in
  if length < 2
  then None
  else let flength = float length in
    let mean = sum /. flength in
    let rec residuals (lst : float list) : float =
    match lst with
    | [] -> 0.
    | hd :: tl -> (hd -. mean) ** 2.
                  +. residuals tl in
    Some (residuals lst /. (flength -. 1.)) ;;
```

There is no clean way to implement the residuals calculation without that auxiliary function. If you find one, let me know!

Algebraic Data Types

There are two forms of data types in OCaml:

1. atomic types: with type constructors like `int` and `bool`
2. composite types: with parametrized type constructors like `'a * 'a` and `'a list`

All composite types, with the exception of functions, are structured as algebraic data types, which are data types built by a combination of conjunction and alternation. The details of conjunction and alternation are explained fully in Chapter 11 of *Abstraction and Design In Computation*.

An algebraic data type is a kind of composite data type built on conjunction and alternation.

Think about the following exercise from Lab 5:

```
(* Define a new type, called "color_label", whose value can
be any of the following options: red, crimson, orange, yellow, green,
blue, indigo, or violet. *)

type color_label =
  | Red
  | Crimson
  | Orange
  | Yellow
  | Green
  | Blue
  | Indigo
  | Violet ;;
```

Exercise 2: What data type is `color_label`?

Since there are a set of alternatives, or variant ways, of building elements of this type, it is also called a **variant type**.

`color_label` is an composite type, variant type, or algebraic data type.

Exercise 3: Define a function, `color_temp`, that identifies a given `color_label` as either a warm color or a cool color, denoted by the strings `"warm"` and `"cool"`.

```
let color_temp (c : color_label) : string =
  match c with
  | Red -> "warm"
  | Crimson -> "warm"
  | Orange -> "warm"
  | Yellow -> "warm"
  | Green -> "cool"
  | Blue -> "cool"
  | Indigo -> "cool"
  | Violet -> "cool" ;;
```

Invariants

An invariant is a condition that is known to be true, but that is not necessarily enforced by the compiler. Assuming an input is a positive number could be an example of an invariant. More often, however, invariants refer to conditions that are guaranteed to stay true in internal manipulation of the concrete representation of an abstract data type. For example, the bignum problem set contains invariants for the representation of zero as a bignum that your code must be careful to always continue to uphold.

Now, let's do some error analysis.

Exercise 4: What would happen if you tried to run the following code after defining color label as above and color as in lab?

```
let my_color : color = Simple Pink ;;
```

You would get an error saying that the constructor `Pink` does not belong to the type color_label.

Exercise 5: What about the following?

```
let convert_to_simple (c : color) : color_label =
  match c with
  | Simple x -> x
  | RGB_ -> Red
  | _ -> Blue ;;
```

You would get a warning saying that the last match case is unused.

Now, we'll put everything to use in a series of problems.

Cumulative Exercises:

You and your friends love poker, but you're all terrible at it. As the computer science specialist of the friend group, you decide to create a program in your favorite language, OCaml, to simulate poker games.

Exercise 6: Define a new type, `card` that stores information about a given playing card. The variants should reflect the difference between a regular playing card, `Regular` and a joker, `Joker`. Think carefully about the information that a card stores, and feel free to define other types to support it.

```
 type card_suit = Heart | Club | Spade | Diamond ;;

type card_name = Ace | King | Queen | Jack | Simple of int ;;

type regular = { suit : card_suit; name : card_name; } ;;

type card = | Card of regular | Joker ;;
```

Exercise 7: Enforce the invariant that a card cannot have a number higher than 10. For example, there is no 11 of spades. Call this `valid_card.`

```
let valid_card (c : card) : bool =
  match c with
  | Card r -> (match r.name with
    | Simple i -> i < 11 && i > 0
    | _ -> true)
  | _ -> true ;;
```

Exercise 8: Your friends are not as proficient in coding for you, so you have to correct some of their mistakes. Convert invalid cards to valid ones if possible. For example, if you receive an invalid card with name 11, convert it to a card with name Jack. The following are the admissible conversions:

- 11 -> Jack
- 12 -> Queen
- 13 -> King
- 14 -> Ace

Otherwise, return None. Call this function `convert_bad_cards`.

```
let convert_bad_cards (c : card) : card option =
  if valid_card c
  then Some c
  else match c with
  | Card r -> (match r.name with
    | Simple i -> (match i with
      | 11 -> Some (Card { suit = r.suit; name = Jack})
      | 12 -> Some (Card { suit = r.suit; name = Queen})
      | 13 -> Some (Card { suit = r.suit; name = King})
      | 14 -> Some (Card { suit = r.suit; name = Ace})
      | _ -> None)
    | _ -> Some c)
  | _ -> Some c ;;
```

Exercise 9: Finally, you're concerned that the root of all your problems has been your friends inputting fake cards into the system when playing games. In order to fix that problem, you decide to write a function `valid_hand` that takes a list of `card` values and returns `true` if they are all valid. Implement that function below.

```
let valid_hand (clst : card list) : bool =
  let valid_list = List.filter (fun x -> valid_card x) clst in
  List.length clst = List.length valid_list ;;
```

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.