Code Review 3 Solutions: Algebraic Types

CS51
Melissa Kwan

1 Design tips

1. Avoid unnecessary match statements and catchall match cases.
2. Get rid of conditionals that result in `true` or `false`.
3. If your helper function is only used in one function, define the helper inside the other function.

2 Defining your own types

2.1 Type synonyms.

A "type synonym" is a new name for an already existing type.

```
type point = float * float ;;
type matrix = float list list ;;
```

Anywhere that a `float * float` is expected, you could use `point`, and vice-versa. The two are completely interchangeable. The function `dist_from_origin` doesn't care whether you pass it a value that is annotated as a `point` versus as a `float * float`. one vs. the other:

```
let dist_from_origin ((x, y) : point) : float =
    sqrt (x ** 2 +. y ** 2) ;;

let pt : point = (3., 4.) ;;
let floatpair : float*float = (3., 4.) ;;

let pt_dist  = dist_from_origin pt ;;
let float_dist = dist_from_origin floatpair ;;
```

2.2 Variants

Here are some examples of constant variants. (Syntax note: These all have to be capitalized, so OCaml knows you're defining a new variant.)

```
type day = Sun | Mon | Tue | Wed
         | Thu | Fri | Sat ;;
type bender = Air | Fire | Water | Earth ;;
type class_year = FirstYear | Sophomore | Junior | Senior | Alum ;;
```

2.3 Records.

You can define a record out of pre-existing types. (Note that I'm using `class_year` from the previous section.)

```
type student = {name : string;
                concentration : string;
                year : class_year;} ;;
let melissa = {name = "Melissa Kwan"; year = Senior} ;;
```

When taking in a record as input, you can use these handy constructs to make your code more readable.

```
(* Bad *)
let label (s : student) : string =
    match p with
    | {concentration = c; year = y; _} ->
        c ^ " " ^ y ;;

(* Using field punning and field selection *)
let label ({concentration; year; _} : student) : string =
    concentration ^ " " ^ year ;;
```

2.4 Algebraic Data Types

Putting all these types together, you get algebraic data types! How to define a variant type:

```
type t =
| C1 [of t1]
| ...
| Cn [of tn]
```

The square brackets above denote the the [of ti] is optional. Every constructor may individually either carry no data or carry data.

Variants also make it possible to discriminate which tag a value was constructed with, even if multiple constructors carry the same type.

```
type t = Left of int | Right of int ;;
let x = Left 1 ;;
let double_right i =
```

```
        match i with
        | Left i -> i
        | Right i -> 2*i ;;
```

Using variants, we can express a type that represents the union of several other types, but in a type-safe way. Here, for example, is a type that represents either a `string` or an `int`: Variants can provide a type-safe way of doing something that might before have seemed impossible.

```
    type string_or_int =
    | String of string
    | Int of int
```

Exercise 2.1. Write a function `sum` that returns the sum of a list of `string_or_int` elements as an integer.

Solution. This problem is a spin on the typical sum problem, which we already know how to do with `fold_left`. The goal is to convert our `string_or_int list` input into a list of integers that we can sum as usual. To do this, we define an `extract_int` function to return the integer value of a `string_or_int` type.

Now we have a way of extracting the integer from each list. We map this function over the list to get a list of integers, and we find integer sum using `fold_left`.

```
let sum (lst : string_or_int list): int =
    let extract_int (x : string_or_int) : int =
        match x with
        | Int i -> i
        | String s ->
            match int_of_string_opt s with
            | Some n -> n
            | None -> raise (Invalid_argument "invalid string argument: not a number")
    in
    let int_lst = List.map extract_int lst in
    List.fold_left (+) 0 int_lst ;;
```

2.5 Parametrized + Recursive variants.

Recursive variants. Variant types may mention their own name inside their own body.

Parametrized variants. In the following example, `bintree` is a "type constructor": there is no concrete way to write a value of type `bintree`. But we can write value of type `int bintree` and `string bintree`. Think of a type constructor as being like a function, but one that maps types to types, rather than values to values.

Binary trees are both parametrized and recursive.

```
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 2.2. Define a function `depth` that takes in a tree with any element type and returns the maximum depth of the tree. The depth of a leaf is 0.

```
(* SOLUTION *)
let rec depth (t : 'a bintree) : int =
  match t with
  | Leaf -> 0
  | Node (_, left, right) -> 1 + max (depth left) (depth right) ;;
```

Exercise 2.3. Define a function `mirror` that takes in a tree with any element type and swaps the left and right branches of every node.

```
(* SOLUTION *)
let rec mirror (t : 'a bintree) : 'a bintree =
  match t with
  | Leaf -> Leaf
  | Node (value, left, right) -> Node (value, mirror right, mirror left) ;;
```

Exercise 2.4. Write a function `fold_tree` that takes in a function `f` of type `'a -> 'b -> 'b -> 'b`, a starting accumulator `acc` of type `'b`, and a binary tree of type `'a bintree`. The function `fold_tree` applies `f` to an element value and the intermediate accumulators of the left and right trees, and outputs a new accumulator. At the very end, `fold_tree` returns the final value of `acc`.

Hint: How is this related to the definitions of `fold` for lists?

```
(* SOLUTION *)
let rec fold_tree (fn : 'a -> 'b -> 'b -> 'b) (acc : 'b) (tree : 'a bintree) : 'b =
  match tree with
  | Leaf -> acc
  | Node (value, left, right) ->
    fn value (fold_tree fn acc left) (fold_tree fn acc right) ;;
```

Exercise 2.5. Write `depth` using `fold_tree`.

```
(* SOLUTION *)
let depth (t : 'a bintree) =
    fold_tree (fun _ acc1 acc2 -> 1 + max acc1 acc2) 0 t ;;
```

Exercise 2.6. Write `mirror` using `fold_tree`.

```
(* SOLUTION *)
let mirror (t : 'a bintree) =
    fold_tree (fun el left right -> Node(el, right, left)) Leaf t ;;
```

Exercise 2.7. Challenge: Write `map_tree` using `fold_tree`.

```
(* SOLUTION *)
let map_tree (f : 'a -> b) (t : 'a bintree) =
    fold_tree (fun el acc1 acc2 -> Node(f el, acc1, acc2) Leaf t ;;
```

3 Advice on Bignums

3.1 Exercise-specific tips

1. `less / greater / equal`: Think about the role of both negatives and the lengths of the respective bignums. You shouldn't need to define a condition for each one.
2. `int_to_bignum, bignum_to_int`: Write out integers in terms of powers of `cBASE`. How can you use `mod` and `/` to break up the bignum?
3. `add`: Think about carrying. How does that factor into `cBASE`?
4. `multiply`: Again, think about powers of `cBASE`. It's helpful to look at an example of the grade-school algorithm for multiplication. The only thing that changed is that `cBASE` is no longer 10; it's 1000. How can you use your intuition from grade school to solve this problem in a similar way?
5. **Design considerations.**

- Make sure you're not spelling out repetitive operations within functions; define those as internal helper functions.
- If your conditional depends on multiple things

3.2 Testing: Utop Alternatives

1. `#use`. Change directories (`cd`) into your lab / problem set folder. Then, from `utop`, you can use `#use "mapfold.ml"` to directly import all your functions.
2. `.makefile`. Create a file called `.makefile` in your problem set directory with commands adjusted based on the template below. Then, you can run `make all` on your command line to build the files, and `./mapfold_tests.byte` to run the tests. Before you submit to Gradescrope, make sure to run `make clean` in your directory to remove all the `byte` files that resulted from the build.

```
all: ps2 ps2_tests

ps2: mapfold.ml
    ocamlbuild -use-ocamlfind mapfold.byte

ps2_tests: mapfold_tests.ml
    ocamlbuild -use-ocamlfind mapfold_tests.byte

clean:
    rm -rf _build *.byte
```