CS 51
CODE REVIEW 3

SAMUEL GREEN AND GABBI MERZ

Contents

Note that there are *no* new syntax here – we're just putting previous concepts and syntax together and thinking about modeling tradeoffs and other abstract concepts!

1. Invariants

Definition 1. *An **invariant** is a condition that is assumed to true of any value of a particular type, but that is not necessarily enforced by the compiler.*

Lab 3 and Problem Set 3 focus significantly on the design and enforcement of invariants as a useful mechanism for programming in OCaml.

What was the invariant of the `RGB` variant of the `color` type in Lab 3?

> **Solution:** The invariant was that the integer values in the Cartesian type were all in the range (0, 255).

Did the choice of the type help us enforce the invariant? What burden does this place on us as programmers? Could a different type definition have helped? Does the inclusion of `color_label` help for any specific reason?

> **Solution:** The type system unfortunately can't do anything more than guarantee that each type is an `int`, which is better than nothing.
>
> We had to write our own validation code to `enforce` this invariant, to make sure that it was always true. As programmers, this is

Date: February 17, 2017.

work for us and for our collaborators and particularly emphasizes the importance of code that is both self-documenting (well styled) and documented (i.e. invariants are recorded in a comment.)

The inclusion of `color_label` is a nice choice because it covers many of the simplest values, meaning that users of the code can construct common values of our `color` type without having enforce the invariant on their own. It's also maintainable, since we can make changes to which red or green we want to use by default very easily!

2. Modeling Choices

Because we have to autograde the labs and problem sets, there doesn't end up being a lot of space to exercise your modeling muscles when it comes to representing data in OCaml. Did you agree with the modeling choices we made in the lab?

Consider for example the following type for color:

```
# type color_label = Orange | Brown | Green ;;
type color_label = Orange | Brown | Green
# type color = Simple of color_label | RGB of int * int * int ;;
type color = Simple of color_label | RGB of int * int * int
```

Why did choose this definition, particularly of `RGB`? What do you think of this redefinition?

```
# type rgb = { r : int ; g : int ; b : int } ;;
type rgb = { r : int; g : int; b : int; }
# type color = { simple : color_label option; rgb : rgb option } ;;
type color = { simple : color_label option; rgb : rgb option; }
```

Is this example better or worse? Is there something better or worse? Is there an example that's similarly "good" to the first version but that uses a record type instead?

> **Solution:** There's a reasonable argument that representing an RGB triple as a record could make sense.

```
# type rgb = { r : int ; g : int ; b : int } ;;
type rgb = { r : int; g : int; b : int; }
# type color = Simple of color_label | RGB of rgb ;;
type color = Simple of color_label | RGB of rgb
```

3. Concise and Field Punning: `valid_date`

Here's a long version of the `valid_date` function. What do you think of its design and style?

```
# let valid_date (d : date) : date =
  if d.year <= 0 then raise (Invalid_Date "only positive years") else
  if d.month = 1 || d.month = 3 || d.month = 5 || d.month = 7 || d.month = 8 ||
     d.month = 10 || d.month = 12 then
    (if d.day > 31 then raise (Invalid_Date "too many days") else
    if d.day < 1 then raise (Invalid_Date "days must be >1") else
    d) else
  if d.month = 4 || d.month = 6 || d.month = 9 || d.month = 11 then
    (if d.day > 30 then raise (Invalid_Date "too many days") else
    if d.day < 1 then raise (Invalid_Date "days must be >1") else
    d) else
  if d.month = 2 then
    (if d.year mod 4 = 0 && d.year mod 100 <> 0 || d.year mod 400 = 0 then
      if d.day > 29 then raise (Invalid_Date "too many days") else
      if d.day < 1 then raise (Invalid_Date "days must be >1") else
      d
    else
      if d.day > 28 then raise (Invalid_Date "too many days") else
      if d.day < 1 then raise (Invalid_Date "days must be >1") else
      d) else
  raise (Invalid_Date "bad month") ;;
```

What are some first observations about this code, perhaps some easy simplifications that could be made?

Solution: Here's a non-exhaustive list of things that come to mind.

- Day, Month, and Year are *all* referenced multiple times throughout this piece of code. Using **field punning** would make a lot of sense here.
- Exception raising logic is repeated – we could extract the raising logic into a separate helper function.
- This is a situation where a `match` makes sense for an `int`, because of there are distinct cases for us to consider.

Here's a potential improved version:

```
# let valid_date ({ year; month; day } as d) : date =
  let raise_bad m = raise (Invalid_Date m) in
  if year < 0 then raise_bad "only positive years"
```

```
else
  let leap = year mod 4 = 0 && year mod 100 <> 0 || year mod 400 = 0 in
  let max_days =
    match month with
    | 1 | 3 | 5 | 7 | 8 | 10 | 12 -> 31
    | 4 | 6 | 9 | 11 -> 30
    | 2 -> if leap then 29 else 28
    | _ -> raise_bad "bad month" in
  if day > max_days then raise_bad "too many days"
  else if day < 1 then raise_bad "days must be >1"
  else d ;;
```

4. Abstraction: count_people and the List module.

Recall the family tree modeling problem from the Lab. (Aside: did you like the model? Did you agree with the type definitions?)

Here's a definition of the function `count_people` from the lab:

```
let rec count_people (fam : family) : int =
  match fam with
  | Single _ -> 1
  | Family (_, _, fl) -> 2 + count_people fl ;;
```

Great. Right? (What's the error here?)

How can we improve this implementation?

> **Solution:** Here's a version that uses recursion directly. This seems a little repetitive, though!

```
# let rec count_people (f : family) : int =
  (* helper function to sum over family list *)
  let rec count_list (l : family list) : int =
    match l with [] -> 0
    | hd :: tl -> (count_people hd) + count_list tl in

  match f with
  Single _ -> 1
  | Family (_, _, fl) -> 2 + count_list fl ;;
```

> **Solution:** I think we can make this even more concise as follows:

```
# let rec count_people (fam : family) : int =
  match fam with
  | Family (_, _, children) ->
    2 + List.fold_left
          (fun prev sub_fam -> count_people sub_fam + prev)
          0 children
  | _ -> 1;;
```

This should be understood as "collapsing" the family tree into a single summary statistic about the whole thing – the magic is that we have a fold within a fold (via the recursive call to `count_people`) that nests a recursion across the list of families *and* down each family in the list at the same time.

5. Nested Recursion: `find_parents`

6. Problem Set 3: Tips 'n' Tricks

- Keep your eyes open for `fold`ing opportunities. They can appear where you least expect them and make for very slick code!
- Built in operators are powerful, and OCaml knows a lot about how to manipulate values. The `Pervasives` module has all of the secrets. For example, do you know the type of the `( = )` function? How about other comparators?

```
# ( = ) ;;
- : 'a -> 'a -> bool = <fun>
```

- Make conscious choices about what method of record field accessing you use. Using the "easiest" can make your code much more complicated than necessary!
- Be deliberate about understanding the `times` function and the algorithm **before** you start to code: the code will only get in the way.
- Appropriately scope helper functions. *If you write a helper function that is only ever called within one other function it should be scoped as such.*
- Make sure to read and understand the style guide.