Code Review 3 Handout

Josh Seides

Topic Outline

This week we are moving from higher-order functions to user-defined values, including variants and algebraic data types.

- variants and invariants
- aside on design, etc.
- algebraic data types
- error handling
- importing and exporting files

Variants and Invariants

Using **variants** is a way to represent complex data in OCaml that is easily usable with pattern matching.

The basic structure for creating variants is to create a **type constructor** similar to a pattern matching case.

Problem 1 *Define a `student` type of which is either a name, GPA, or enrolled boolean which can be used for identification (but not really).*

To actually use variants after defining them, they can be called with a **value constructor** similar to defining `option` types.

Problem 2 *Create a few students of type `student`.*

The main benefit to using variants is that they can easily be pattern matched on in a similar way to **deconstructing** other data types like lists or tuples.

Problem 3 *Implement a function that extracts the GPA float to a `student` value if possible.*

What is the difference between a variant and an **invariant**? An invariant is simply an assumption in the code that must be maintained throughout the execution of a program.

Aside on Design, etc.

Consider the following implementation for `valid_rgb` from lab.

```
let valid_rgb color =
    let bad_color c = c < 0 || c > 255 in
    match color with
    | Simple color -> Simple color
    | RGB (r, g, b) ->
        if bad_color r then raise (Invalid_Color "red out of range")
        else if bad_color g then raise (Invalid_Color "green int out of range")
        else if bad_color b then raise (Invalid_Color "blue int out of range")
        else color ;;
```

Problem 4 *How can the implementation above be improved in terms of design?*

Again.

```
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") ;;
```

Problem 5 *How can the implementation above be improved in terms of design?*

Some general things I saw last week * repeated `match` cases * single-element `match` cases * single-case `match` cases * opportunities to condense `match` cases with `_` or input names * extraneous parentheses * `true` and `false` in `if` statements * `=` vs. `==` and `<>` vs. `!=` * general spacing concerns * `@` vs. `::`

Algebraic Data Types

ADTs are a general term used to describe data types that include variants, records, and tuples. ADTs have similar benefits to variants.

Look at the staff solution to `valid_date`

```
let valid_date ({year;month;day} as d) : date =
    if year < 0 then raise (Invalid_Date "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 (Invalid_Date "bad month") in
    if day > max_days then raise (Invalid_Date "too many days")
    else if day < 1 then raise (Invalid_Date "days must be >1")
    else d ;;
```

Note especially * pattern matching in the input directly * field punning in the input definition * reference to the entire input as one object * multiple `match` cases syntax * `&&` and `||`

Problem 6 *Recall from lab the `family` type. Implement `marry` and `add_to_family` from lab.*

```
type person = { name : string; favorite : color; birthdate : date; } ;;

(type family =
    | Child of person
    | Family of person * person * family list) ;;
```

Problem 7 *Implement `count_people` from lab.*

Let's try something a little different. This is tougher conceptually but very interesting and useful (!).

Problem 8 *How would we define an `'a` binary tree as an ADT?*

Problem 9 *How would we count the size of our `'a` binary tree? The height?*

```
let t = Br(2, Br (1, Lf, Lf), Br(3, Lf, Lf)) ;;

size t ;;

height t ;;
```

Error Handling

There are two ways to handle errors: **options** and **exceptions**.

Problem 10 *What are the differences and pros/cons of using each alternative for error handling?*

Problem 11 *What are the types of the following?*

```
Some 42 ;;
[None] ;;
Failure "rip" ;;
raise (Failure "rip") ;;
raise ;;
fun _ -> raise Exit ;;
```

Importing and Exporting Files

Important things to look into * `open ...` * `let open ... in` * `#use ...`