Code Review 2: February 14, 2020

Currying, Records and Tuples, and Anomalous Conditions

Hakeem Angulu, hangulu@college.harvard.edu

Learning Goals

- Describe the differences between a curried and an uncurried function
- Understand partial application and when to use it
- Compare records and tuples to understand the use-cases of both
- Be able to handle some anomalous conditions with option types

---

Logistical Notes

Problem Set 2 due Sunday - Office hours on Saturday and Sunday - Map/Fold + Higher order functions

Problem Set 3 due next Saturday - More complicated custom types - Additional office hours on Wednesday and Thursday

---

Currying and Partial Application

All functions in OCaml take exactly one argument.

This is an incredibly powerful and useful point, and makes use of the fact that functions are first-order objects in OCaml.

Let's look at what exactly we mean when we say that all functions in OCaml take exactly one argument.

Example:

```
let rec power (n, x) =
  if n = 0 then 1
  else x * power ((n - 1), x) ;;
```

The function power takes an integer and raises it to a given power. Note that it takes **one** argument here, instead of several, and that argument is a **tuple**.

But, what you may think is straightforward syntax is actual syntactic sugar for currying. We can desugar the above to get:

```
let rec power =
  fun (n, x) ->
    if n = 0 then 1
    else x * power ((n - 1), x) ;;
```

But once again, this is even more syntactic sugar, with pattern matching in the function's argument. If we remove it, we see the true identity of the `power` function:

```
let rec power =
  fun argument ->
    match argument with
    | (n, x) -> if n = 0 then 1
                else x * power ((n - 1), x) ;;
```

With this, we see that power was always a function of one argument.

How about a power function that takes the base and the exponent as two arguments, instead of as a tuple? I've implemented it below:

```
let rec power n x =
  if n = 0 then 1
  else x * power (n - 1) x ;;
```

Exercise 1: Using the same process of desugaring, unveil the true identity of this power function. That is, rewrite it such that the idea that all functions in OCaml take exactly one argument is clear.

```
let rec power =
  failwith "power not implemented" ;;
```

Partial application is explained in detail on page 98 of our textbook,*Abstraction in Design and Computation*, but the following is a quick intuitive explanation and review.

Partial application is defined as:**the applying of a curried function to only some of its arguments, resulting in a function that takes the remaining arguments.**

We use the term “currying” to mean:**encoding a multi-argument function using nested, higher-order functions.** In OCaml, we tend to use curried functions, rather than uncurried definitions. We saw currying at the beginning of Lab 4.

Currying can be a bit confusing, so a CS51 TF from a few years ago, Sundar Solai, came up with the metaphor of thinking of a curried function as actual spicy curry -- something so hot and spicy that you have to take it in in small bites.

Hence, a function is curried**if it takes in it’s arguments in small bites, i.e one at a time**A function is uncurried **if you can take in all the arguments in one big gulp.**

With that said, let's look at the infamous Lab 4 functions `curry` and `uncurry`.

As with all functions you're asked to write in this class, you should first think of what these functions should do on a fundamental level.

`curry` should take in an uncurried function (i.e. a function that takes in its parameters in a single tuple -- the big gulp) and allow it to accept input **as if it were a curried function**-- which means, with the inputs given in small bites -- then return its usual result.

`uncurry` should take in a curried function (i.e. a function that takes in its parameters one at a time -- the small bites) and allow it to accept input **as if it were an uncurried function**-- which means, with the inputs given all at once -- then return its usual result.

If you're currying a function, you want it to take in its parameters as small bites, and if you're uncurrying a function, you want it take in its parameters as a tuple.

With that said, given the confusion in Lab 4, we will now reimplement `curry` and `uncurry` as an exercise:

Exercise 2: Define polymorphic higher-order functions curry and uncurry for currying and uncurrying binary functions (functions of two arguments).

```
let curry =
  fun _ -> failwith "curry not implemented" ;;

let uncurry =
  fun _ -> failwith "uncurry not implemented" ;;
```

Let's clarify partial application before moving on. An intuitive explanation of partial application is as follows:

The applying of a curried function to only some of its arguments, resulting in a function that takes the remaining arguments.

The order in which a curried function takes its arguments thus becomes an important design consideration, as it determines what partial applications are possible.

Let's continue with the power function.

Exercise 3: If you gave the following power function one argument in utop, what would OCaml return?

```
let rec power n x =
  if n = 0 then 1
  else x * power (n - 1) x ;;
```

```
>> power 3 ;;
-: _
```

Exercise 4: Use the power function, with partial application, to write a function that cubes every element in a list.

```
let cube_list =
  failwith "cube_list not implemented" ;;
```

Exercise 5: Use even more partial application with map to write a function that seemingly takes no arguments, but cubes every element in a list. Do not modify the function's type signature below. (Hint: think about how you would call the cube_list function above in utop)

```
let cube_list2 =
  failwith "cube_list2 not implemented" ;;
```

Partial application and currying are very powerful and the essence of idiomatic OCaml. Using them in your labs and problem sets will net you many more design points.

Records and Tuples

A record is a user-defined data type that allows one to name the elements of a set with unique labels.

Take the following example from our upcoming Lab 5 (sneak peak!):

```
(* Define a person record type. Use the field names "name",
"favorite", and "birthdate". *)

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

The record above stores data of the type `person` with the **fields**: `name`, `favorite`, and `birthdate`. The fields are labeled, so their order does not matter. This may be reminiscent of the `dict` data type in Python, or the `Map` interface in Java.

Exercise 6: Construct a value (replacing `yourname` with your first name) of the `person` type above that represents yourself.

```
let yourname = _ ;;
```

Exercise 7: Define a function, `fave_color`, that generates a sentence like "Hakeem Angulu's favorite color is black," replacing "Hakeem Angulu" and "black" with your name and favorite color from the value in Exercise 6.

```
let fave_color (p : person) : string =
  failwith "fave_color not implemented" ;;
```

Like with many problems in this class, many solutions to Exercise 7 exist. Below are a few ways that you can manipulate records in OCaml:

- Field punning: when matching record patterns, for fields in which the label and the variable name are identical, the variable name may be omitted. For example, instead of writing:

```
match p with
| {name = n; color = c; _} -> ...
```

one can write:

```
match p wth
| {name; color; _} -> ...
```

- Single-pattern matching in the let construct: one can eliminate the explicit `match` entirely by pattern matching within the let construct as such:

```
let fave_color ({name; color; _} : person) =
  ...
```

where instead of `...`, one access the fields of the `person` type with `name` and `color` as the variable names.

- Field selection: much like accessing attributes in a Python object, like a `dict`, one can access the fields of a record in OCaml with dot notation as such:

```
let fave_color (p : person) =
  ...
```

where instead of `...`, one can access the fields of the `person` type with `p.name`, `p.color`, and `p.birthdate`.

The record is conceptually very similar to the tuple: they both allow for the composition of different types into one value. However, as has been shown, records are very explicit about what each field is, and thus could be deemed preferable to tuples in all cases.

However, there are a few cases in which records may be unnecessary, and tuples may suffice. Take, for example, the traditional representation of coordinates:

```
 (* Defined with a tuple *)
type coords_tup = int * int ;;

(* Defined with a record *)
type coords_rec = {latitude : int; longitude : int} ;;
```

While the record version is certainly more explicit with what each number in the ordering represents, that information is often deemed unnecessary. The well-known convention is that when two numbers are given as coordinates, the first represents latitude and the second represents longitude, so explicit labeling may be

unnecessary and thus constitute poor design.

Exercise 8: Define two types to store colors in RGB format, `color_tup` and `color_rec`, that are a tuple and record respectively. Which would you prefer and why?

```
type color_tup = _ ;;

type color_rec = _ ;;
```

Anomalous Conditions

Sometimes, functions get inputs that they cannot handle well. In these situations, which we call **anomalous conditions**, there are two things that we can do:

1. Use option types
2. Raise an exception

We'll continue going over exceptions in later material, so for this code review, we'll focus on **the option type.**

In short, options provide a way of dealing with a value that might or might not be there. They are built by putting the prefix `Some` in front of a value of another type. For example, if we put `Some 5` in utop, OCaml would tell us the type was as follows:

```
>> Some 5
:- int option = Some 5
```

The other option for option types is `None`. Let's try that in utop as well:

```
>> None
:- 'a option = None
```

Hopefully Lab 4 gave you adequate practice with these types, so let's look at an example:

```
let rec max_list (lst: int list) : int option =
  match lst with
  | [] -> None
  | head :: tail ->
     match (max_list tail) with
     | None -> Some head
     | Some max_tail -> Some (max head max_tail) ;;
```

`max_list` takes an `int list` and returns its maximum. We've seen this function before, but we've struggled with how to define the maximum of an empty list. Now, we've handled that anomalous condition (that is, the inputted list `lst` being empty), with the option type, matching that case with `None`.

Possibly the biggest issue that students face is deconstructing options, as is done above in the second `match` statement. This is critically important, because it allows us to access the values within the option type. You **almost always** need to pattern match an option type to do something useful with it.

Exercise 9: Write a function, `zero_ints`, that takes in an `int option` and returns an `int`, or `0` if `None` is given.

```
let zero_ints (x : int option) : int =
  failwith "zero_ints not implemented" ;;
```

The solutions to all of the above exercises will be available on Sunday at 10am to give you some time to work through them again on your own.

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.