O’Caml Basics: Options

COS 326
David Walker
Princeton University
Options

A value $v$ has type $t$ option if it is either:

- the value $\text{None}$, or
- a value $\text{Some } v'$, and $v'$ has type $t$

Options can signal there is no useful result to the computation

Example: we look up a value in a hash table using a key.

- If the key is present, return $\text{Some } v$ where $v$ is the associated value
- If the key is not present, we return $\text{None}$
Slope between two points

type point = float * float

let slope (p1:point) (p2:point) : float =
Slope between two points

type point = float * float

let slope (p1:point) (p2:point) : float =
  let (x1,y1) = p1 in
  let (x2,y2) = p2 in
  ;;
  deconstruct tuple
Slope between two points

type point = float * float

let slope (p1:point) (p2:point) : float =
  let (x1,y1) = p1 in
  let (x2,y2) = p2 in
  let xd = x2 -. x1 in
  if xd != 0.0 then
    (y2 -. y1) /. xd
  else
    ???
;;

avoid divide by zero

what can we return?
Slope between two points

```ocaml
type point = float * float

let slope (p1:point) (p2:point) : float option =
    let (x1,y1) = p1 in
    let (x2,y2) = p2 in
    let xd = x2 -. x1 in
    if xd != 0.0 then
        ???
    else
        ???
    ;;
```

we need an option type as the result type
type point = float * float

let slope (p1:point) (p2:point) : float option =
  let (x1,y1) = p1 in
  let (x2,y2) = p2 in
  let xd = x2 -. x1 in
  if xd != 0.0 then
    Some ((y2 -. y1) /. xd)
  else
    None
;
type point = float * float

let slope (p1:point) (p2:point) : float option =
let (x1,y1) = p1 in
let (x2,y2) = p2 in
let xd = x2 -. x1 in
if xd != 0.0 then
  (y2 -. y1) /. xd
else
  None
;;
Slope between two points

```ml
type point = float * float

let slope (p1:point) (p2:point) : float option =
  let (x1,y1) = p1 in
  let (x2,y2) = p2 in
  let xd = x2 -. x1 in
  if xd != 0.0 then
    (y2 -. y1) /. xd
  else
    None
```

- **type point** = float * float
- **let slope** (p1:point) (p2:point) : float option =
  - let (x1,y1) = p1 in
  - let (x2,y2) = p2 in
  - let xd = x2 -. x1 in
  - if xd != 0.0 then
    - (y2 -. y1) /. xd
  - else
    - None

**Has type float**

**Can have type float option**

**WRONG: Type mismatch**
Slope between two points

```ocaml
type point = float * float

let slope (p1:point) (p2:point) : float option =
  let (x1,y1) = p1 in
  let (x2,y2) = p2 in
  let xd = x2 -. x1 in
  if xd != 0.0 then
    (y2 -. y1) /. xd
  else
    None
;;
```
doubly WRONG: result does not match declared result

Has type float
Remember the typing rule for `if`

if \( e_1 : \text{bool} \)
and \( e_2 : t \) and \( e_3 : t \) (for some type \( t \))
then if \( e_1 \) then \( e_2 \) else \( e_3 : t \)

- Returning an optional value from an if statement:

if ... then

\[
\begin{align*}
\text{None} & : t \text{ option} \\
\text{Some ( ... )} & : t \text{ option}
\end{align*}
\]
How do we use an option?

slope : point -> point -> float option

returns a float option
How do we use an option?

```ocaml
slopes : point -> point -> float option

let print_slope (p1:point) (p2:point) : unit =
```

;;
How do we use an option?

```
slope : point -> point -> float option

let print_slope (p1:point) (p2:point) : unit =
  slope p1 p2

;;
```

returns a float option; to print we must discover if it is None or Some
How do we use an option?

```ocaml
slopes : point -> point -> float option

let print_slope (p1:point) (p2:point) : unit =
  match slope p1 p2 with
  ;;
```
How do we use an option?

slope : point -> point -> float option

let print_slope (p1:point) (p2:point) : unit =
  match slope p1 p2 with
  | Some s ->
  | None ->
  ;;

There are two possibilities

Vertical bar separates possibilities
How do we use an option?

slope : point -> point -> float option

let print_slope (p1:point) (p2:point) : unit =
    match slope p1 p2 with
    Some s ->
    | None ->
    ;;

The "Some s" pattern includes the variable s

The object between | and -> is called a pattern
How do we use an option?

```ocaml
slopexpoint -> point -> float option

let print_slope (p1:point) (p2:point) : unit =
  match slope p1 p2 with
  | Some s ->
      print_string ("Slope: " ^ string_of_float s)
  | None ->
      print_string "Vertical line.\n"
;;
```
Writing Functions Over Typed Data

• Steps to writing functions over typed data:
  1. Write down the function and argument names
  2. Write down argument and result types
  3. Write down some examples (in a comment)
  4. **Deconstruct** input data structures
  5. **Build** new output values
  6. Clean up by identifying repeated patterns

• For option types:

  when the **input** has type `t option`,
  deconstruct with:

  ```
  match ... with
  | None -> ...
  | Some s -> ...
  ```

  when the **output** has type `t option`,
  construct with:

  ```
  Some (...)
  None
  ```
MORE PATTERN MATCHING
type point = float * float

let distance (p1:point) (p2:point) : float =
  let square x = x *. x in
  let (x1,y1) = p1 in
  let (x2,y2) = p2 in
  sqrt (square (x2 -. x1) +. square (y2 -. y1))
Recall the Distance Function

```ocaml
type point = float * float

let distance (p1:point) (p2:point) : float =
  let square x = x *. x in
  let (x1,y1) = p1 in
  let (x2,y2) = p2 in
  sqrt (square (x2 -. x1) +. square (y2 -. y1))
;;
```

(x2, y2) is an example of a pattern – a pattern for tuples.

So let declarations can contain patterns just like match statements.

The difference is that a match allows you to consider multiple different data shapes.
Recall the Distance Function

type point = float * float

let distance (p1:point) (p2:point) : float =
  let square x = x *. x in
  match p1 with
  | (x1,y1) ->
    let (x2,y2) = p2 in
    sqrt (square (x2 -. x1) +. square (y2 -. y1))
;;

There is only 1 possibility when matching a pair
Recall the Distance Function

type point = float * float

let distance (p1:point) (p2:point) : float =
    let square x = x *. x in
    match p1 with
    | (x1,y1) ->
        match p2 with
        | (x2,y2) ->
            sqrt (square (x2 -. x1) +. square (y2 -. y1))

We can nest one match expression inside another.
(We can nest any expression inside any other, if the expressions have the right types)
we built a pair of pairs

let distance (p1:point) (p2:point) : float =
  let square x = x *. x in
  match (p1, p2) with
  | ((x1,y1), (x2, y2)) ->
    sqrt (square (x2 -. x1) +. square (y2 -. y1))

Pattern for a pair of pairs:  ((variable, variable), (variable, variable))
All the variable names in the pattern must be different.
A pattern must be consistent with the type of the expression in between `match ... with`.

We use `(p3, p4)` here instead of `((x1, y1), (x2, y2))`
I like the original the best

```
type point = float * float

let distance (p1:point) (p2:point) : float =
  let square x = x *. x in
  let (x1,y1) = p1 in
  let (x2,y2) = p2 in
  sqrt (square (x2 -. x1) +. square (y2 -. y1))
;;
```

It is the clearest and most compact.
Code with unnecessary nested patterns matching is particularly ugly to read.
You'll be judged on code style in this class.
type point = float * float

(* returns a nearby point in the graph if one exists *)
nearby : graph -> point -> point option

let printer (g:graph) (p:point) : unit =
  match nearby g p with
  | None -> print_string "could not find one\n"
  | Some (x,y) ->
    print_float x;
    print_string ", ";
    print_float y;
    print_newline();
;;
• Constant values can be used as patterns

```ocaml
let small_prime (n:int) : bool =
  match n with
  | 2 -> true
  | 3 -> true
  | 5 -> true
  | _ -> false
;;
```

```ocaml
let iffy (b:bool) : int =
  match b with
  | true -> 0
  | false -> 1
;;
```

the underscore pattern matches anything
it is the "don't care" pattern
A SHORT JAVA RANT
Definition and Use of Java Pairs

public class Pair {
    public int x;
    public int y;

    public Pair (int a, int b) {
        x = a;
        y = b;
    }
}

public class User {
    public Pair swap (Pair p1) {
        Pair p2 =
            new Pair(p1.y, p1.x);
        return p2;
    }
}

What could go wrong?
The input p1 to swap may be null and we forgot to check.

Java has no way to define a pair data structure that is just a pair.

How many students in the class have seen an accidental null pointer exception thrown in their Java code?
In O'Caml, if a pair may be null it is a pair option:

```ocaml
type java_pair = (int * int) option
```
In O'Caml, if a pair may be null it is a pair option:

```ocaml
type java_pair = (int * int) option
```

And if you write code like this:

```ocaml
let swap_java_pair (p:java_pair) : java_pair =
  let (x,y) = p in
  (y,x)
```
From Java Pairs to O'Caml Pairs

In O'Caml, if a pair may be null it is a pair option:

```ocaml
type java_pair = (int * int) option
```

And if you write code like this:

```ocaml
let swap_java_pair (p:java_pair) : java_pair =
  let (x,y) = p in
  (y,x)
```

You get a **helpful** error message like this:

```ocaml
# ... Characters 91-92:
  let (x,y) = p in (y,x);;
  ^
Error: This expression has type java_pair = (int * int) option
    but an expression was expected of type 'a * 'b
```
From Java Pairs to O'Caml Pairs

type java_pair = (int * int) option

And what if you were up at 3am trying to finish your COS 326 assignment and you accidentally wrote the following sleep-deprived, brain-dead statement?

let swap_java_pair (p:java_pair) : java_pair =
  match p with
  | Some (x,y) -> Some (y,x)
From Java Pairs to O'Caml Pairs

type java_pair = (int * int) option

And what if you were up at 3am trying to finish your COS 326 assignment and you accidentally wrote the following sleep-deprived, brain-dead statement?

let swap_java_pair (p:java_pair) : java_pair =
  match p with
  | Some (x,y) -> Some (y,x)

OCaml to the rescue!

..match p with
  | Some (x,y) -> Some (y,x)

Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched: None
From Java Pairs to O'Caml Pairs

```ocaml
type java_pair = (int * int) option

let swap_java_pair (p:java_pair) : java_pair =
  match p with
  | Some (x,y) -> Some (y,x)
  | None -> None
```

And what if you were up at 3am trying to finish your COS 326 assignment and you accidentally wrote the following sleep-deprived, brain-dead statement?

```ocaml
let swap_java_pair (p:java_pair) : java_pair =
  match p with
  | Some (x,y) -> Some (y,x)
```

An easy fix!
Moreover, your pairs are probably almost never null!

Defensive programming & always checking for null is annoying.
From Java Pairs to O'Caml Pairs

There just isn't always some "good thing" for a function to do when it receives a bad input, like a null pointer.

In O'Caml, all these issues disappear when you use the proper type for a pair and that type contains no "extra junk."

```
type pair = int * int
```

Once you know O'Caml, it is hard to write swap incorrectly.
Your bullet-proof code is much simpler than in Java.

```
let swap (p:pair) : pair =
  let (x,y) = p in (y,x)
```
Java has a paucity of types
   – There is no type to describe just the pairs
   – There is no type to describe just the triples
   – There is no type to describe the pairs of pairs
   – There is no type ...

OCaml has many more types
   – use option when things may be null
   – do not use option when things are not null
   – OCaml types describe data structures more precisely
     • programmers have fewer cases to worry about
     • entire classes of errors just go away
     • type checking and pattern analysis help prevent programmers from ever forgetting about a case
Java has a paucity of types

- There is no type to describe just the pairs
- There is no type to describe just the triples
- There is no type to describe the pairs of pairs
- There is no type...

OCaml has many more types

- Use op.on when things may be null
- Do not use op.on when things are not null
- Ocaml types describe data structures more precisely

• Programmers have fewer cases to worry about
• Errors just go away
• Type checking and pattern analysis help prevent programmers from ever forgetting about a case

SCORE: OCAML 1, JAVA 0
OVERALL SUMMARY:
A SHORT INTRODUCTION TO
FUNCTIONAL PROGRAMMING
Functional Programming

Steps to writing functions over typed data:

1. Write down the function and argument names
2. Write down argument and result types
3. Write down some examples
4. Deconstruct input data structures
   • the argument types suggest how you do it
   • the types tell you which cases you must cover
5. Build new output values
   • the result type suggests how you do it
6. Clean up by identifying repeated patterns
   • define and reuse helper functions
   • refactor code to use your helpers
   • your code should be elegant and easy to read
### Summary: Constructing/Deconstructing Values

<table>
<thead>
<tr>
<th>Type</th>
<th>Construct Values</th>
<th>Number of Cases</th>
<th>Deconstruct Values</th>
</tr>
</thead>
</table>
| int     | 0, -1, 2, ...    | $2^{31}-1$      | match i with 
|         |                  |                  |   | 0 -> ...
|         |                  |                  |   | -1 -> ...
|         |                  |                  |   | ... |
|         |                  |                  |   | x -> ...
| bool    | true, false      | 2               | match b with 
|         |                  |                  |   | true -> ...
|         |                  |                  |   | false -> .... |
| t1 * t2 | (2, "hi")       | (# of t1) * (# of t2) | let (x,y) = ... in ...
|         |                  |                  | match p with (x,y) -> ...
| unit    | ()               | 1               | e1; ...
| t option| None, Some 3     | 1 + (# of t1)   | match opt with 
|         |                  |                  |   | None -> ...
|         |                  |                  |   | Some x -> ...


END