OCaml: Tuples and Higher-Order Functions

Programming Languages

William Killian

Millersville University

Outline

- Tuples
  - Syntax
  - Bindings
  - Pattern Matching
- Higher-Order Functions
  - Definition
  - Anonymous Functions
- Bonus: Bindings <==> Anonymous Functions

Tuples

Tuples

- Tuples are a ***product type***
- Used for when we want to group entities together
- Elements are access by <u>location</u>

```
type student = string * int * float
```

- We created a new type called student
- It is an <u>*alias*</u> (or another name for a tuple)
- This tuple contains a string, an int, and a float

Tuple Syntax

- How could we store a point?
- What is its datatype (as a tuple)?
- How can we create a new point?

Tuple Syntax

- How could we store a point?
  - We should be able to store a point as a pair of coordinates
  - We can access its data by “location”
- What is its datatype (as a tuple)?

```
type point = float * float
```

  This means that a point is modeled as two floats

- How can we create a new point?

```
let my_point = (1.2, 0.0)
let my_point : point = (1.2, 0.0)
let my_point : float * float = (1.2, 0.0)
(* all three of these are the same! *)
```

Tuple Syntax

Expression / Value:

```
("Dr. Killian", 327291, 3.38)
```

Type:

```
string * int * float
```

Always enclosed in parentheses

Datatypes can be deduced for each element

Immutable – you cannot change a tuple

- You can read from a tuple
- You can create a new tuple

Tuple Bindings

Binding refresher: providing a name to a value

```
let point = (2.0, 3.14)
```

Extracting the “x” value of the point:

```
let (x, _) = point
```

Extracting the “y” value of the point:

```
let (_, y) = point
```

Note: The _ means to ignore

Tuple Bindings

```
let big = (1, 3.14, "hello", true, 5)
```

1. What is the type of big?

2. How can we extract the 2nd, 4th, and 5th elements with identifiers "pi", "passing", and "courses" ?

3. How can we compare the 1st and 5th element for equality? (hint: two steps)

Tuple Bindings

```
let big = (1, 3.14, "hello", true, 5)
```

1. What is the type of big?

   ```
   int * float * string * bool * int
   ```

2. How can we extract the 2nd, 4th, and 5th elements with identifiers "pi", "passing", and "courses" ?

   ```
   let (_, pi, _, passing, courses) = big
   ```

3. How can we compare the 1st and 5th element for equality? (hint: two steps)

   ```
   let eq = let (first, _, _, _, last) in
                first = last
   ```

Pattern Matching

- Tuples can lend to clean, expressive code when combined with pattern matching
- Can be combined with other patterns (e.g. for lists)

Problem: Compute the centroid (geometric average) of three points which form a triangle.

```
let points = [(0.0, 1.0),
              (6.0, 2.0),
              (3.0, 5.0)]
```

What is the type of points?

Pattern Matching Examples

Normal List:

```
match l with
| [] -> (* empty list *)
| h::t -> (* have more *)
```

Normal Tuple:

```
match p with
| (0,0) -> (* origin *)
| (x,y) -> (* general point *)
```

Centroid

```
let centroid lst =
  let rec average sum n lst =
    match lst with
    | [] ->
        let (x, y) = sum in (* pull out each coordinate *)
          (x /. n, y /. n) (* compute average *)
    | (x,y)::lst’ ->
        (* pull out each coordinate *)
        let (xs, ys) = sum in (* evolve arguments *)
          average (x +. xs, y +. ys) (n +. 1.0) lst'
  in
  average 0.0 0.0 lst (* sum=0.0, n=0.0 *)
```

Pattern Matching Problem

- Count the number of origin points in a list

```
let rec count_origin lst =
```

Pattern Matching Problem

- Count the number of origin points in a list

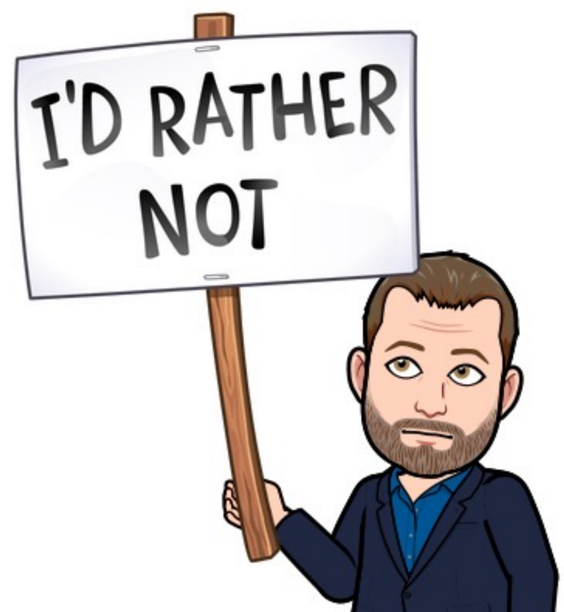
```
let rec count_origin lst =
  match lst with
  | (0,0)::lst' -> (1 + count_origin lst')
  | _::lst' -> count_origin lst'
```

Higher Order Functions

Higher Order Functions (HOFs)

- **Functions that either**
  - Accept one (or more) functions as parameters
  - Return a function as a result

- Functions accepting functions as parameters?
- Functions returning functions?



Why Use Higher-Order Functions?

- Composition
  - We can first create smaller functions that solve simple problems
  - Then we can compose them together to solve complex problems

- Reduces bugs
- Improves readability
- Enables generic programming / reuse

Example: map

We have already written one HOF: `map`

```
let rec map f l =
  match l with
  | [] -> []
  | h::t -> (f h)::(map f l)
```

```
f       : 'a -> 'b
l       : 'a list
returns : 'b list
```

Without map...

```
let rec map_float_of_int l =
  match l with
  | [] -> []
  | h::t ->
    (float_of_int h)::(map_float_of_int l)

let rec map_string_of_float l =
  match l with
  | [] -> []
  | h::t ->
    (string_of_float h)::(map_string_of_float l)
```

With map…

```
let rec map f l =
  match l with
  | [] -> []
  | h::t -> (f h)::(map f l)

let map_float_of_int l =
  map float_of_int l

let map_string_of_float l =
  map string_of_float l
```

A More Complex Example

Given a list of integers, I want to:

1. Convert them to a float
2. Then convert the floats to a string

Essentially:

```
data → float_of_int → string_of_float

[1;2;3] → [1.0;2.0;3.0] → ["1.0";"2.0";"3.0"]
```

A More Complex Example

```
let complex l =
  map string_of_float (map float_of_int l)

let complex l =
  map (fun x -> string_of_float (float_of_int x)) l
```

- Both are equivalent in what they do
- The top must call **map** twice
- The bottom must call **map** only once

fun – a function by no-name

We usually write bindings as:

```
let add x y = x + y
```

But we can write:

```
let add = fun x y -> x + y
```

`fun` is used to indicate that we have a function

- But this function has no name.
- This is called an anonymous (or *lambda*) function

Revisiting the Complex Example

```
let complex l =
  map string_of_float (map float_of_int l)
```

```
let complex l =
  map (fun x -> string_of_float (float_of_int x)) l
```

Now if only we could get rid of some of these parens…

```
let complex l =
  l |> map float_of_int |> map string_of_float
```

```
let complex l =
  map (fun x -> float_of_int x |> string_of_float)
l
```

The Pipeline Operator |>

- Probably one of the coolest functions ever(?)
- Super short definition:

```
let (|>) a f = f a
```

- Swaps the position of the first argument with the function name. This is known as a "data-first" pattern
- This means the function's first argument comes before the |> operator
- Evaluation now "in-order" left-to-right

The Pipeline Operator in Use

```
[-1.2; 1.0; 0.5; 3.5; -5.5; 0.75; 4.2; 0.31]

let magic (l:float list) = l
  |> List.filter (fun x -> x >= 0.0)
  |> List.filter (fun x -> x <= 1.0)
  |> List.map (fun x -> x * 100.0)
  |> List.map int_of_float
  |> List.map string_of_int
  |> List.map (fun x -> x ^ " ")
     (* string concatenation *)
  |> List.fold_left (^) ""
```

The Pipeline Operator **not** in Use

```
[-1.2; 1.0; 0.5; 3.5; -5.5; 0.75; 4.2; 0.31]

let magic (l:float list) = l
 List.fold_left (^) ""
 (List.map (fun x -> x ^ " ")
 (List.map string_of_int
 (List.map int_of_float
 (List.map (fun x -> x * 100.0)
 (List.filter (fun x -> x <= 1.0)
 (List.filter (fun x -> x >= 0.0)
 l))))))
```

Revisiting Bindings

```
let x = e in expr
```

can be rewritten as:

```
(fun x -> expr) (e)
```

In fact, it's what the interpreter does!

```
let x = 5 in
let y = x * 2 in
  x + y
```

Revisiting Bindings

```
let x = 5 in
let y = x * 2 in
  x + y

(fun x ->
let y = x * 2 in
  x + y
) (5)

(fun x ->
(fun y ->
  x + y) (x * 2)
) (5)
```