OCaml: Higher-Order Functions

Programming Languages

William Killian

Millersville University

Outline

- Higher-Order Functions
  - Definition
  - Anonymous Functions
- Bonus: Bindings <==> Anonymous 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

`data → float_of_int → string_of_float`

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…

Remember, we want to emulate the following:

```
data → float_of_int → string_of_float
```

Revisiting the Complex Example

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 (Local) 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 Local Bindings - Trace

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

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

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

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

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

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