OCaml: Folding

Programming Languages

William Killian

Millersville University
OCaml Tour

• Types, Values, and Expressions
• Bindings
• Strong Typing + Type Inference
• Functions + Tail Recursion
• Pattern Matching
• Lists, Tuples, Strings
• Variants + Recursive Types
• Folding ← We are now here
List Operations

Three Major Classes

• Transforming elements
  map
• Removing elements
  filter
• Combining elements
  fold
List Operations: `map`

**Type Signature**

\( (\text{'a} \to \text{'b}) \to \text{'a list} \to \text{'b list} \)

`function` `list` `return`

**Usage**

`List.map fn lst`

**Description**

Calls `fn` on each element of `lst`.

\[ [ \text{fn x1}; \text{fn x2}; \text{fn x3}; \ldots ] \]
map examples

List.map (fun x -> x*x) [1; 2; 3; 4; 5]

List.map string_of_float [3.14; 2.78; 3.30]

List.map (fun (a,b) -> (b,a)) [(2,1);(3,4);(5,6)]
List Operations: **filter**

Type Signature

\[(\text{'a} \rightarrow \text{bool}) \rightarrow \text{'a list} \rightarrow \text{'a list}\]

**predicate**       **list**       **return**

Usage

`List.filter pred lst`

Description

Calls `fn` on each element of `lst`, only keeping elements who satisfy the predicate.
filter examples

List.filter (fun x -> x mod 2 <> 0) [1; 2; 3; 4]

List.filter (fun x -> x > 10) [1; 330; 2020; 30]

List.filter (function Some _ -> true | _ -> false)
    [None; Some 5; None; None; Some 2; Some 1]
List Operations: **fold_left**

**Type Signature**

\[
('a \rightarrow 'b \rightarrow 'a) \rightarrow 'a \rightarrow 'b \ \text{list} \rightarrow 'a
\]

**Usage**

```haskell
List.fold_left \( \text{fn} \ \text{init} \ \text{list} \ \text{return} \)
```

**Description**

Calls combiner on init + each element

\[
\text{fn} \ ( \ldots \ \text{fn} \ (\text{fn} \ \text{init} \ b1) \ b2 \ \ldots \ bN)
\]
fold_left examples

List.fold_left (+) 0 [1; 2; 3; 4]

List.fold_left (^) "" ["a"; "b"; "c"; "d"]

List.fold_left (fun acc x -> x :: acc) [] [1; 2; 3; 4]
List Operations: **fold_right**

**Type Signature**

\((\text{'a} \rightarrow \text{'b} \rightarrow \text{'b}) \rightarrow \text{'a list} \rightarrow \text{'b} \rightarrow \text{'b}\)

**Usage**

`List.fold_right fn a init`

**Description**

Calls combiner on `init` + each element

`fn a1 (fn a2 ( ... fn aN b ))`
fold_right examples

List.fold_right (+) [1; 2; 3; 4] 0

List.fold_right (^) ["a"; "b"; "c"; "d"] ""

List.fold_right (List.cons) [1; 2; 3; 4] []
List module (subset 1/2)

**rev** : `'a list` -> `'a list`
Reverse a list (Tail Recursive)

**concat** : `'a list list` -> `'a list`
Concatenate list of lists

**map** : (`'a` -> `'b`) -> `'a list` -> `'b list`
Apply function to each element

**mapi** : (`int` -> `'a` -> `'b`) -> `'a list` -> `'b list`
Same as above (with index) (Tail Recursive)

**rev_map** : (`'a` -> `'b`) -> `'a list` -> `'b list`
Same as map, but reverses output (Tail Recursive)
List module (subset 2/2)

`iter : ('a -> unit) -> 'a list -> unit`
Call a function on each element (*Tail Recursive*)

`iteri : (int -> 'a -> unit) -> 'a list -> unit`
Same as above (with index) (*Tail Recursive*)

`mem : 'a -> 'a list -> bool`
Search for value in a list (*Tail Recursive*)

`filter : ('a -> bool) -> 'a list -> 'a list`
Returns a list with all elements that satisfy the predicate
Everything is a Fold
List.rev is a fold_left

let rev lst =
  let combiner acc x =
    x :: acc
  in
  let init = []
  in
  List.fold_left combiner init lst
List.concat is a fold_left

```ocaml
let concat lst =
  let combiner = acc x =
    acc @ x
  in
  let init = []
  in
  List.fold_left combiner init lst
```
List.map is a fold_right

let map fn lst =
    let combiner x acc =
        fn x :: acc
    in
    let init = []
    in
    List.fold_right combiner lst init
List.mapi is a fold_right

```plaintext
let mapi fn lst =
    let combiner x (i, acc) =
        (i + 1, fn i x :: acc)
    in
    let init = (0, [])
    in
    snd (List.fold_right combiner lst init)
```
List.rev_map is a fold_left

let rev_map fn lst =
  let combiner acc x =
    fn x :: acc
  in
  let init = []
  in
  List.fold_left combiner init lst
List.iter is a fold_left

```ocaml
let iter fn lst =
  let combiner acc x =
    fn x
  in
  let init = ()
  in
  List.fold_left combiner init lst
```
List.iteri is a fold_left

```ocaml
define iteri (
  fn : (int -> int)
  lst : list
) =
  let combiner (i, acc) x =
    (i + 1, fn i x)
  in
  let init = (0, ())
  in
  snd (List.fold_left combiner init lst)
```

List\_mem is a fold\_left

```plaintext
let mem value lst =
  let combiner acc x =
    if acc then acc else value = x
  in
  let init = false
  in
  List.fold\_left combiner init lst
```
List.filter is a fold_right

let filter pred lst =
  let combiner x acc =
    if pred x then x :: acc else acc
  in
  let init = []
  in
  List.fold_right combiner lst init
fold_right is a fold_left
fold_right could be a fold_left

let fold_right fn lst init =
  let lst' = rev lst
  in
  let fn' acc x = fn x acc
  in
  fold_left fn' init lst'

How can we implement `fold_left` and `fold_right`?
fold_left

let rec fold_left fn init lst =
fold_left

let rec fold_left fn init lst =
  match lst with
  | [] -> init
  | x::lst' ->
    fold_left fn (fn init x) lst'
fold_right

let rec fold_right fn lst init =
fold_right

let rec fold_right fn lst init =
  match lst with
  | [] -> init
  | x::lst' ->
    fn x (fold_right fn lst' init)
OCaml Tour

- Types, Values, and Expressions
- Bindings
- Strong Typing + Type Inference
- Functions + Tail Recursion
- Pattern Matching
- Lists, Tuples, Strings
- Variants + Recursive Types
- Folding ← We were here
OCaml Tour

You are *Functional Programming Masters*

Only **ONE** more lab
Fin

So long and thanks for all the : :