CS 3110

Lecture 6: Map and Fold

Prof. Clarkson
Spring 2015

Today’s music: Selections from the soundtrack to 2001: A Space Odyssey
Review

Course so far:
• Syntax and semantics of (most of) OCaml

Today:
• No new language features
• New idioms:
  – Map, fold, and other higher-order functions
Question #1

How much of PS1 have you finished?

A. None
B. About 25%
C. About 50%
D. About 75%
E. I’m done!!!
Review: higher-order functions

• Functions are values

• Can use them anywhere we use values
  – Arguments, results, parts of tuples, bound to variables, carried by datatype constructors or exceptions, ...

• First-class citizens of language, afforded all the “rights” of any other values
  – Functions can take functions as arguments
  – Functions can return functions as results
    ...functions can be higher-order
Review: anonymous functions

(aka function expressions)

• Syntax:

- fun x -> e
- fun p -> e

• Type checking:
  - Conclude that fun x -> e : t1 -> t2
    if e : t2 under assumption x : t1

• Evaluation:
  - A function is already a value
Lambda

- In PL, anonymous functions a.k.a. *lambda expressions* \( \lambda x . e \)
- The lambda means “what follows is an anonymous function”
  - \( x \) is its argument
  - \( e \) is its body
  - Just like \( \text{fun } x \rightarrow e \), but slightly different syntax
- Standard feature of any functional language (ML, Haskell, Scheme, ...)

- You’ll see “lambda” show up in many places in PL, e.g.:
  - Lambda style: [https://www.youtube.com/watch?v=Ci48kqp11F8](https://www.youtube.com/watch?v=Ci48kqp11F8)
Recall: every OCaml function takes exactly one argument

– Can encode $n$ arguments with one $n$-tuple

– Or, can write function that takes one argument and returns a function that takes another argument and...

– Called “currying” after famous logician Haskell Curry vs.
Haskell B. Curry

Languages Haskell and Curry named for him

Curry-Howard isomorphism:
• Types are logical formulas
• Programs are logical proofs

\[ \text{fun } x \rightarrow x : \texttt{'}a \rightarrow \texttt{'}a \]

1900-1982
HUGE HIGHER-ORDER FUNCTIONS

Discovery of the monolith:
https://www.youtube.com/watch?v=ML1OZCHixR0
Map

bad style!

\[
\text{map } (\text{fun } x \to \text{shirt\_color}(x)) \quad [\quad ]
\]

= [\text{gold, blue, red}]
Map

map shirt_color [ ]

= [gold, blue, red]

how would you implement map?
Map

let rec map f xs =
  match xs with
  | [] -> []
  | x::xs' -> (f x)::(map f xs')

map : ('a -> 'b) -> 'a list -> 'b list

Map is HUGE:
- You use it all the time once you know it
- Exists in standard library as List.map, but the idea can be used in any data structure (trees, stacks, queues...)

let rec map f xs =
  match xs with
  | [] -> []
  | x::xs' -> (f x)::(map f xs')

map : ('a -> 'b) -> 'a list -> 'b list
Question #2

What is value of lst after this code?

```
let is_even x = (x mod 2 = 0)
let lst = map is_even [1;2;3;4]
```

A. [1;2;3;4]
B. [2;4]
C. [false; true; false; true]
D. false
E. None of the above
Question #2

What is value of lst after this code?

```plaintext
let is_even x = (x mod 2 = 0)
let lst = map is_even [1;2;3;4]
```

A. [1;2;3;4]
B. [2;4]
C. [false; true; false; true]
D. false
E. None of the above
Filter

filter is_vulcan [ ] = [ ]

(er, half vulcan)

how would you implement filter?
Filter

let rec filter f xs =
  match xs with
  | [] -> []
  | x::xs' -> if f x
    then x::(filter f xs')
    else filter f xs'

filter : (a -> bool) -> 'a list -> 'a list

Filter is also HUGE
  – In library: List.filter
Question #3

What is value of lst after this code?

```plaintext
let is_even x = (x mod 2 = 0)
let lst = filter is_even [1;2;3;4]
```

A. [1;2;3;4]
B. [2;4]
C. [false; true; false; true]
D. false
What is value of lst after this code?

```plaintext
let is_even x = (x mod 2 = 0)
let lst = filter is_even [1;2;3;4]
```

A. [1;2;3;4]
B. [2;4]
C. [false; true; false; true]
D. false
Iterators

• Map and filter are *iterators*
  – Not built-in to the language, an **idiom**

• Benefit of iterators: separate recursive traversal from data processing
  – Can reuse same traversal for different data processing
  – Can reuse same data processing for different data structures
  – leads to modular, maintainable, beautiful code!

• So far: iterators that change or omit data
  – what about combining data?
  – e.g., sum all elements of list
**Fold v1.0**

Idea: *stick an operator between every element of list*

\[
\text{folding } [1; 2; 3] \quad \text{with} \quad (+) \\
\text{becomes} \\
1 + 2 + 3 \\
\quad \Rightarrow * \\
6
\]
Fold v2.0

Idea: *stick an operator between every element of list*

*But list could have 1 element, so need an initial value*

folding \([1]\)  with 0 and (+)

becomes

\[
0+1 \\
\rightarrow \ast \\
1
\]
Fold v2.0

Idea: *stick an operator between every element of list*
*But list could have 1 element, so need an initial value*

folding $[1;2;3]$ with 0 and (+)
becomes
$0+1+2+3$

$\rightarrow*$

$6$
Fold v2.0

Idea: *stick an operator between every element of list*
*But list could have 1 element, so need an initial value*
*Or list could be empty; just return initial value*

folding \([\texttt{[]}\) with \(0\) and \((+\)\) becomes

\(0\)
Question #4

What should the result of folding $[1;2;3;4]$ with 1 and $(\ast\ast)$ be?

A. 1  
B. 24  
C. 10  
D. 0  
E. None of the above
Question #4

What should the result of folding \([1;2;3;4]\) with 1 and \((\ast\ast)\) be?

A. 1
B. 24
C. 10
D. 0
E. None of the above
Fold v3.0

Idea: *stick an operator between every element of list*
But list could have 1 element, so need an initial value
Or list could be empty; just return initial value

Implementation detail: iterate left-to-right or right-to-left?

folding \([1;2;3]\) with 0 and (+)

left to right becomes: \(((0+1)+2)+3\) \(\text{...fold_left}\)
right to left becomes: \(1+(2+(3+0))\) \(\text{...fold_right}\)

Both evaluate to 6; does it matter?

Yes: not all operators are *associative*
e.g. subtraction, division, exponentiation, ...
Question #5

Result of \texttt{fold\_right} with input list \([1;2;3]\), initial accumulator \(1\) and operator \((-)\)?

A. -5  
B. -1  
C. 1  
D. Operator has the wrong type  
E. None of the above
Question #5

Result of `fold_right` with input list `[1;2;3]`, initial accumulator 1 and operator (−)?

A. -5
B. -1
C. 1
D. Operator has the wrong type
E. None of the above
Question #6

Result of `fold_left` with input list `[1;2;3]`, initial accumulator 1 and operator `(-)`?

A. -5
B. -1
C. 1
D. Operator has the wrong type
E. None of the above
Question #6

Result of \texttt{fold\_left} with input list \([1;2;3]\), initial accumulator 1 and operator \((-)\)?

A. -5
B. -1
C. 1
D. Operator has the wrong type
E. None of the above
Fold v4.0

• (+) *accumulated* a result of the same type as list itself
• What about operators that change the type?
  – e.g., let cons x xs = x :: xs
    cons : 'a -> 'a list -> 'a list
    folding from the right \([1;2;3]\) with \([]\) and cons
    should produce
    \(1::(2::(3::[])) = [1;2;3]\)
• So the operator needs to accept
  – the accumulated result so far, and
  – the next element of the input list
    ...
    which may have different types!
Fold for real

Two versions in OCaml library:

```ocaml
List.fold_left
: ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
```

```ocaml
List.fold_right
: ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
```
Fold for real

Two versions in OCaml library:

```ocaml
List.fold_left
: ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
```

```ocaml
List.fold_right
: ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
```

Operator

mnemonic: operator takes in accumulator on the (left|right)
Fold for real

Two versions in OCaml library:

List.fold_left
: ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a

List.fold_right
: ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b

Input list
Fold for real

Two versions in OCaml library:

```ocaml
List.fold_left
: ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
```

```ocaml
List.fold_right
: ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
```

Initial value of accumulator

mnemonic: fold takes in accumulator on the (left|right) of the list
Fold for real

Two versions in OCaml library:

```ocaml
List.fold_left
  : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
```

```ocaml
List.fold_right
  : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
```

Final value of accumulator
fold_left

let rec fold_left f acc xs =
    match xs with
    | [] -> acc
    | x::xs' -> fold_left f (f acc x) xs'

Accumulates an answer by
• repeatedly applying \( f \) to “answer so far”,
• starting with initial value \( \text{acc} \),
• folding “from the left”

fold_left \( f \) \( \text{acc} \) \([a;b;c]\)
computes
\( f (f (f \text{acc} a) b) c \)
fold_right

let rec fold_right f xs acc =
  match xs with
    []       -> acc
  | x::xs'   -> f x (fold_right f xs' acc)

Accumulates an answer by
• repeatedly applying \( f \) to “answer so far”,
• starting with initial value \( \text{acc} \),
• folding “from the right”

fold_right f [a;b;c] acc
computes
\( f \ a \ (f \ b \ (f \ c \ \text{acc})) \)
Behold the HUGE power of fold

Implement so many other functions with fold!

let rev xs = fold_left (fun xs x -> x::xs) [] xs
let length xs = fold_left (fun a _ -> a+1) 0 xs
let map f xs = fold_right
    (fun x a -> (f x)::a) xs []
let filter f xs = fold_right
    (fun x a -> if f x then x::a else a) xs []
Beware the efficiency of fold

- Implementation of `fold_left` more space efficient than `fold_right` for long lists
  - what is "long"? maybe 10,000 elements
- But that doesn’t mean that one is strictly better than the other
- More in recitation...
Map-Reduce

• Fold has many synonyms/cousins in various functional languages, including `scan` and `reduce`
• Google organizes large-scale data-parallel computations with Map-Reduce
  — open source implementation by Apache called Hadoop

“[Google’s Map-Reduce] abstraction is inspired by the map and reduce primitives present in Lisp and many other functional languages. We realized that most of our computations involved applying a map operation to each logical record in our input in order to compute a set of intermediate key/value pairs, and then applying a reduce operation to all the values that shared the same key in order to combine the derived data appropriately.”
[Dean and Ghemawat, 2008]