CSCI 2041: First Class Functions

Chris Kauffman Chris Kauffman

Last Updated:
Wed Oct 10 08:40:40 CDT 2018
Logistics

Reading

- OCaml System Manual: Ch 26: List and Array Modules, higher-order functions
- Practical OCaml: Ch 8

Goals

- Functions as parameters
- Higher-order Functions
- Map / Reduce / Filter
- Functions as return values
- Currying

Exam 1

- Grades Up on Gradescope
- Regrade requests via Gradescope, Due Mon 10/15

Assignment 3 multimanager

- Manage multiple lists
- Records to track lists/undo
- option to deal with editing
- Higher-order funcs for easy bulk operations
- Post by Tue
- Due Mon 10/22
Exercise: Code Patterns on Lists

1. Describe the code structure that they share
2. Describe which parts differ between them
3. What is the shared purpose of the functions

```
let rec evens list = (* all even ints in list *)
  match list with
  | [] -> []
  | h::t when h mod 2 = 0 -> h::(evens t)
  | _::t -> evens t
  ;;

let rec shorter lim list = (* all strings shorter than lim *)
  match list with
  | [] -> []
  | h::t when String.length h < lim -> h::(shorter lim t)
  | _::t -> shorter lim t
  ;;

let rec betwixt min max list = (* elements between min/max *)
  match list with
  | [] -> []
  | h::t when min<h && h<max -> h::(betwixt min max t)
  | _::t -> betwixt min max t
```
Answers: Code Patterns on Lists

1. Describe the code structure that they share
   - Each destructures the list and examines which elements satisfy some criteria.
   - List of the "true" elements results while "false" elements are excluded.

2. Describe which parts differ between them
   - The specific criteria for each function differs: evenness, string length, and within a range
   - The parameters associated with these conditions also change

3. What is the shared purpose of the functions
   - To filter a list down to elements for which some condition is true

Identifying a code pattern that is mostly copy-pasted creates an opportunity to write less and get more. OCaml provides a means to encapsulate this code pattern and others.
Functions as Parameters

- OCaml features *1st class functions*
  - Functions can be passed as parameters to other functions
  - Functions can be returned as values from functions
  - Functions can be bound to names just as other values, global, local, or mutable names

- **Higher-order function**: function which takes other functions as parameters, i.e. a function OF functions

- Many code patterns can be encapsulated via higher-order functions
Exercise: Basic Examples of Higher-Order Functions

Determine values bound to a, b, c

```ocaml
(* Higher-order function which applies func as a function to arg. *)
let apply func arg = func arg ;;

(* Simple arithmetic functions. *)
let incr n = n+1;;
let double n = 2*n;;

let a = apply incr 5;;
let b = apply double 5;;
let c = apply List.hd ["p";"q";"r"]
```

Determine values bound to x, y, z

```ocaml
(* Higher-order function taking two function parameters f1 and f2. Applies them in succession to arg. *)
let apply_both f1 f2 arg = let res1 = f1 arg in let res12 = f2 res1 in res12 ;;

let x = apply_both incr double 10;;
let y = apply_both double incr 10;;
let z = apply_both List.tl List.hd ["p";"q";"r"];;
```

Determine the types for the two higher-order functions apply and apply_both shown below.
Answers: Basic Examples of Higher-Order Functions

\[
a = \text{apply incr 5} \\
\quad = (\text{incr 5}) \\
\quad = 6 \\

b = \text{apply double 5} \\
\quad = (\text{double 5}) \\
\quad = 10 \\

c = \text{apply List.hd ["p";"q";"r"]} \\
\quad = \text{List.hd ["p";"q";"r"]} \\
\quad = "p" \\

x = \text{apply_both incr double 10} \\
\quad = (\text{double (incr 10)}) \\
\quad = (\text{double 11}) \\
\quad = 22 \\

y = \text{apply_both double incr 10} \\
\quad = (\text{incr (double 10)}) \\
\quad = (\text{incr 20}) \\
\quad = 21 \\

z = \text{apply_both List.tl List.hd ["p";"q";"r"]} \\
\quad = (\text{List.hd (List.tl ["p";"q";"r"]))} \\
\quad = (\text{List.hd ["q";"r"]}) \\
\quad = "q"
\]

Function types:

\[
\text{let apply func arg = ...} \\
\text{val apply : ('a -> 'b) -> 'a -> 'b} \\
\quad |--func--| \quad \text{arg} \quad \text{return} \\
\text{let apply_both f1 f2 arg = ...} \\
\text{val apply_both : ('a -> 'b) -> ('b -> 'c) -> 'a -> 'c} \\
\quad |---f1---| \quad |---f2---| \quad \text{arg} \quad \text{return} \\
\]

Note that \text{apply_both} applies param func f1 first then applies f2 to that that result
Exercise: Notation for Function Types

- Fill in the ??? entries in the table below dealing with types
- Entries deal with function param and return types
- Lower entries are higher-order functions
- Be able to describe in words what each entry means

<table>
<thead>
<tr>
<th>Type Notation</th>
<th>#args</th>
<th>arg types</th>
<th>Return Type</th>
<th>Higher Order?</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 int</td>
<td>0</td>
<td>Not a function</td>
<td>int</td>
<td>No</td>
</tr>
<tr>
<td>2 int -&gt; string</td>
<td>1</td>
<td>???</td>
<td>string</td>
<td>No</td>
</tr>
<tr>
<td>3 int -&gt; string -&gt; int</td>
<td>2</td>
<td>??? + ???</td>
<td>???</td>
<td>No</td>
</tr>
<tr>
<td>4 ??? -&gt; bool</td>
<td>3</td>
<td>int + string + int</td>
<td>bool</td>
<td>No</td>
</tr>
<tr>
<td>5 (int -&gt; string) -&gt; int</td>
<td>1</td>
<td>(int -&gt; string)</td>
<td>???</td>
<td>Yes</td>
</tr>
<tr>
<td>6 (int -&gt; string) -&gt; int -&gt; bool</td>
<td>???</td>
<td>???</td>
<td>bool</td>
<td>Yes</td>
</tr>
<tr>
<td>7 ???</td>
<td>2</td>
<td>int + (string -&gt; int)</td>
<td>bool</td>
<td>Yes</td>
</tr>
<tr>
<td>8 (int -&gt; string -&gt; int) -&gt; bool</td>
<td>???</td>
<td>???</td>
<td>bool</td>
<td>Yes</td>
</tr>
</tbody>
</table>
**Answers: Notation for Function Types**

<table>
<thead>
<tr>
<th>Type Notation</th>
<th>#args</th>
<th>arg types</th>
<th>Return Type</th>
<th>Higher Order?</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 int</td>
<td>0</td>
<td>Not a function</td>
<td>int</td>
<td>No</td>
</tr>
<tr>
<td>2 int -&gt; string</td>
<td>1</td>
<td>int</td>
<td>string</td>
<td>No</td>
</tr>
<tr>
<td>3 int -&gt; string -&gt; int</td>
<td>2</td>
<td>int + string</td>
<td>int</td>
<td>No</td>
</tr>
<tr>
<td>4 int -&gt; string -&gt; int -&gt; bool</td>
<td>3</td>
<td>int + string + int</td>
<td>bool</td>
<td>No</td>
</tr>
<tr>
<td>5 (int -&gt; string) -&gt; int</td>
<td>1</td>
<td>(int -&gt; string)</td>
<td>int</td>
<td>Yes</td>
</tr>
<tr>
<td>6 (int -&gt; string) -&gt; int -&gt; bool</td>
<td>2</td>
<td>(int -&gt; string) + int</td>
<td>bool</td>
<td>Yes</td>
</tr>
<tr>
<td>7 int -&gt; (string -&gt; int) -&gt; bool</td>
<td>2</td>
<td>int + (string-&gt; int)</td>
<td>bool</td>
<td>Yes</td>
</tr>
<tr>
<td>8 (int -&gt; string -&gt; int) -&gt; bool</td>
<td>1</td>
<td>(int -&gt; string-&gt; int)</td>
<td>bool</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**What about returning a function?**

- Natural to wonder about type for returning a function. A good guess would be something like
  
  `int -> (string -> int)`
  
  for 1 int param and returning a (string -> int) function

- Will find that this instead written as
  
  `int -> string -> int`
  
  due to OCaml’s **curried functions** (more later)
Filtering as a Higher-order Function

The following function captures the earlier code pattern

(* val filter : ('a -> bool) -> 'a list -> 'a list
  Higher-order function: pred is a function of a single element that
  returns a true/false value, often referred to as a "predicate".
  filter returns a all elements from list for which pred is true *)

let rec filter pred list =
  match list with
  | [] -> []
  | h::t when (pred h)=true -> h::(filter pred t)
  | _::t -> filter pred t

Allows expression of filtering functions using predicates

let evens list = (* even numbers *)
  let is_even n = n mod 2 = 0 in (* predicate: true for even ints *)
  filter is_even list (* call to filter with predicate *)

let shorter lim list = (* strings shortenr than lim *)
  let short s = (String.length s) < lim in (* predicate *)
  filter short list (* call to filter *)

let betwixt min max list = (* elements between min/max *)
  let betw e = min < e && e < max in (* predicate *)
  filter betw list (* call to filter with predicate *)
Exercise: Use filter

- Define equivalent versions of the following functions
- Make use of filter in your solution

(* More functions that filter elements *)
let rec ordered list = (* first pair elem < second *)
  match list with
  | [] -> []
  | (a,b)::t when a < b -> (a,b)::(ordered t)
  | _::t -> ordered t
  ;;

let rec is_some list = (* options that have some *)
  match list with
  | [] -> []
  | (Some a)::t -> (Some a)::(is_some t)
  | _::t -> is_some t
  ;;
Answers: Use filter

(* Definitions using filter higher-order function *)
let ordered list = (* first pair elem < second *)
  let pred (a,b) = a < b in
  filter pred list

let is_some list = (* options that have some *)
  let pred opt = (* named predicate with *)
    match opt with (* formatted source code *)
      | Some a -> true (* that is boring but easy *)
      | None   -> false (* on the eyes *)
    in
  filter pred list

;;
fun with Lambda Expressions

- OCaml’s `fun` syntax allows one to "create" a function
- This function has no name and is referred to alternatively as
  - An anonymous function (e.g. no name)
  - A lambda expression (e.g. many Lisps use keyword lambda instead of `fun` to create functions)
- Lambda (Greek letter \( \lambda \)) was used by Alonzo Church to represent "abstractions" (e.g. functions) in his calculus

```ocaml
let add1_stand x = (* standard function syntax: add1_normal is *)
  let xp1 = x+1 in (* parameterized on x and remains unevaluated *)
  xp1 (* until x is given a concrete value *)
;;

let add1_lambda = (* bind the name add1_lambda to ... *)
  (fun x -> (* a function of 1 parameter named x. *)
   let xp1 = x+1 in (* Above standard syntax is "syntactic sugar" *)
   xp1) (* for the "fun" version. *)
;;

let eight = add1_stand 7;; (* both versions of the function *)
let ate = add1_lambda 7;; (* behave identically *)
```

13
Many higher-order functions require short, one-off function arguments for which `fun` can be useful:

```ocaml
let evens list = (* even numbers *)
    filter (fun n -> n mod 2 = 0) list

let shorter lim list = (* strings shorter than lim *)
    filter (fun s -> (String.length s) < lim) list

let betwixt min max list = (* elements between min/max *)
    filter (fun e -> min < e && e < max) list
```

If predicates are more than a couple lines, favor a named helper function with nicely formatted source code: **readability**

```ocaml
let is_some list = (* options that have some *)
    let pred opt = (* named predicate with *)
        match opt with (* formatted source code *)
        | Some a -> true (* that is boring but easy *)
        | None    -> false (* on the eyes *)
        in
    filter pred list

let is_some list = (* magnificent one-liner version... *)
    filter (fun opt -> match opt with Some a->true | None->false) list
```

First Class Functions Mean *fun* Everywhere

- *fun* most often associated with args to higher-order functions like `filter` BUT...
- A *fun* / lambda expression can be used anywhere a value is expected including but not limited to:
  - Top-level `let` bindings
  - Local `let/in` bindings
  - Elements of a arrays, lists, tuples
  - Values referred to by `refs`
  - Fields of records
- `lambda_expr.ml` demonstrates many of these
- Poke around in this file for a few minutes to see things like...

```ml
(* Demo function refs *)
let func_ref = ref (fun s -> s^" "^s);; (* a ref to a function *)
let bambam = !func_ref "bam";;               (* call the ref’d function *)
func_ref := (fun s -> "!!!");;              (* assign to new function *)
let exclaim = !func_ref "bam";;             (* call the newly ref’d func *)
```
Families of Higher-Order Functions

- Along with `filter`, there are several other common use patterns on data structures
- Most functional languages provide higher-order functions in their standard library for these use patterns on their built-in Data Structures (DS)
- Will discuss each of these: to harness the power of functional programming means **getting intimate with all of them**

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Description</th>
<th>Library Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Filter</td>
<td>Select some elements from a DS</td>
<td>List.filter, Array.filter</td>
</tr>
<tr>
<td></td>
<td>(‘a -&gt; bool) -&gt; ’a DS -&gt; ’a DS</td>
<td></td>
</tr>
<tr>
<td>Iterate</td>
<td>Perform side-effects on each element of a DS</td>
<td>List.iter, Array.iter</td>
</tr>
<tr>
<td></td>
<td>(‘a -&gt; unit) -&gt; ’a DS -&gt; unit</td>
<td>Queue.iter</td>
</tr>
<tr>
<td>Map</td>
<td>Create a new DS with different elements, same size</td>
<td>List.map, Array.map</td>
</tr>
<tr>
<td></td>
<td>(‘a -&gt; ’b) -&gt; ’a DS -&gt; ’b DS</td>
<td></td>
</tr>
<tr>
<td>Reduce/Fold</td>
<td>Compute single value based on all DS elements</td>
<td>List.fold_left / fold_right</td>
</tr>
<tr>
<td></td>
<td>(‘a -&gt; ’b -&gt; ‘a) -&gt; ’a -&gt; ’b DS -&gt; ’a</td>
<td>Array.fold_left / fold_right</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Queue.fold</td>
</tr>
</tbody>
</table>