CSCI 2041: Functions, Mutation, and Arrays

Chris Kauffman

Last Updated:
Fri Sep 14 15:06:04 CDT 2018
Logistics

- OCaml System Manual: 1.1 - 1.3
- Practical OCaml: Ch 1-2
- OCaml System Manual: 25.2 (Pervasives Modules)
- Practical OCaml: Ch 3, 9

Goals Today

- Function Definitions
- Mutation and Arrays
- Polymorphism with Functions

Friday: Lists/Recursion

Lab01

- Submit/Checkoff by next Monday
- How did it go?

Assignment 1

- Due Monday 9/17
- Note a few updates announced on Piazza / Changelog
- Questions?
Exercise: Function Definitions and Types

- Have seen this several times: functions can be defined by binding a name with parameters.
- Functions always have a type that gives their parameters and return type.
- Notation for this in ML is with "arrows" like these examples:
  - `int -> float` (* 1 int param, return float *)
  - `int -> int -> float` (* 2 int params, return float *)
  - `string -> int -> unit` (* string and int params, return nothing *)

What are the types of the following functions?

```ml
(* func_types.ml : func defs / types *)

let do_math x y = (* do some math *)
  let z = x + y in
  let w = z*z + z in
  w
;;

let do_english s = (* make a word *)
  let suffix = "-alicious" in
  s^suffix
;;

open Printf;;
(* Alternate printing strings *)
let repeat_alt_print n str1 str2 =
  for i=1 to n do
    if i mod 2 = 1 then
      printf "%s\n" str1
    else
      printf "%s\n" str2
  done;
;;
```
Answers: Function Definitions and Types

(* func_types.ml : func defs / types *)

let do_math x y = (* do some math *)
  let z = x + y in
  let w = z*z + z in
  w
;;

let do_english s = (* make a word *)
  let suffix = "-alicious" in
  s^suffix
;;

open Printf;;
(* Alternate printing strings *)
let repeat_alt_print n str1 str2 =
  for i=1 to n do
    if i mod 2 = 1 then
      printf "%s\n" str1
    else
      printf "%s\n" str2
  done;
;;

Invoking the compiler as
ocamlc -i will show the inferred types associated with top-level bindings like functions.

> ocamlc -i func_types.ml
val do_math : int -> int -> int
val do_english : string -> string
val repeat_alt_print :
  int -> string -> string -> unit
Annotating Function Types

- For clarity, may annotate functions with their types
- Sometimes hard to tell types of arguments without some clues given in documentation or annotation

(* func_types.annotated.ml : func defs with explicit type annotations *)
open Printf;;

(* Annotate only the arguments *)
let do_math (x : int) (y : int) =
  let z = x + y in
  let w = z*z + z in
  w;;

(* Annotate args and function return *)
let do_english (s : string) : string =
  let suffix = "-alicious" in
  s^suffix;;

(* Annotate args and function return *)
let repeat_alt_print (n:int) (str1:string) (str2:string) : unit =
  for i=1 to n do
    if i mod 2 = 1 then
      printf "%s
" str1
    else
      printf "%s
" str2
  done;;
for/do Loops

- Quite limited compared to C/Java/Python
- Count only up by 1’s or down by 1’s in an integer range
- Last statement of loop gives is the value of the loop expression
- In practice mostly loops have side-effects: unit value
- Focus in most cases is on recursion instead

(* print the first n even numbers *)
let print_evens1 n =
    for i=0 to n-1 do
        let e = 2*i in
        printf "%d : %d\n" i e;
    done;
;;

(* print first n even numbers, descending order *)
let print_evens_descend n =
    for i=n-1 downto 0 do
        let e = 2*i in
        printf "%d : %d\n" i e;
    done;
;;

while/do loops are also available, usually used with refs
if/then/else and Conditional Execution

- if/then/else allows for conditional evaluation
- Usually need both if/else cases as the expression has a value
- When side-effects are intended, only the if portion is required

```ocaml
let is_even n =
  if n mod 2 = 0 then (* mod is remainder operator *)
    true (* return true *)
  else (* return false *)
    false
;;

(* print a message only if even *)
let print_if_even n =
  if is_even n then
    printf "%d is even\n" n; (* no associated else case *)
;;
```
**Exercise:** if/then/else has value

Contrast the two uses of if/then/else below and describe how they are used differently

(* form a string based on even/oddness *)
let even_odd_str1 n =
  if n mod 2 = 0 then
    let nstr = string_of_int n in
    let msg = " is even" in
    nstr^msg
  else
    let nstr = string_of_int n in
    let msg = " is odd" in
    nstr^msg

(* same result, different style *)
let even_odd_str2 n =
  let nstr = string_of_int n in
  let msg =
    if n mod 2 = 0 then
      " is even"
    else
      " is odd"
  in
  nstr^msg

;;
**Answers: if/then/else has value**

- even_odd_str2 exploits binds msg based on a condition
- More abundant in functional languages than imperative

```ml
(* form a string based on even/oddness *)
let even_odd_str1 n =
  (* standard style *)
  if n mod 2 = 0 then
    (* condition with *)
    let nstr = string_of_int n in (* differing assignments *)
    let msg = " is even" in
    nstr^msg
  else
    let nstr = string_of_int n in
    let msg = " is odd" in
    nstr^msg

;;

(* form a string based on even/oddness *)
let even_odd_str2 n = (* more functional style *)
  (* unconditional binding *)
  let msg = (* bind this value.. *)
  if n mod 2 = 0 then (* based on this condition *)
    " is even" (* condition true *)
  else (* condition false *)
    " is odd"
  in
  nstr^msg

;;
```
Refs and Mutation

- Mutable bindings are often done via **references**
- These are set up to "point" at a mutable data location
- Initialize with `ref x` with `x` as the initial value
- Alter the location with ref assignment syntax `x := y;`
- Retrieve ref data with `!x`

(* ref_summing.ml : demonstrate use of mutable refs to sum *)

```ml
open Printf;;

let sum_1_to_n n = (* generate the sum of numbers 1 to n *)
  let sum = ref 0 in (* initialize ref to 0 *)
  for i=1 to n do (* loop *)
    let next = !sum + i in (* add on i to current sum *)
    sum := next; (* assign sum to next; RETURN TYPE unit *)
    (* sum := !sum + i; *) (* above two lines as a one-liner *)
  done;
  !sum (* return value of sum *)
;;
```

```ml
let sum10 = sum_1_to_n 10 in
let sum50 = sum_1_to_n 50 in
printf "summing 1 to 10 gives %d\n" sum10;
printf "summing 1 to 50 gives %d\n" sum50;
;;
```
Exercise: Common Errors involving Refs

The following two are common bugs involving refs/functions that use refs

Explain the two bugs and how to fix them

1 (* ref_errors.ml : contains two errors involving refs *)
2 let ipow x n = (* calculate x to the nth power *)
3   let p = ref 1 in
4   for i=1 to n do
5       p := p * x;
6   done;
7   p
8 ;;
9 (* File "ref_errors.ml", line 5, characters 9-10: 
10   Error: This expression has type int ref 
11   but an expression was expected of type int *)
12
13 let sum = (ipow 2 5) + (ipow 3 7);;
14 (* File "ref_errors.ml", line 13, characters 10-20: 
15   Error: This expression has type int ref 
16   but an expression was expected of type int *)
17
18 Printf.printf "sum is %d\n" sum;;
**Answers: Common Errors involving Refs**

- Both errors involve dereferencing with the `!` operator
- First error: can only add `int`, not `int ref`
- Second error: initially inferred type of the function as `int -> int -> int ref` which is not intended

```ml
1 (* ref_errors_fixed.ml : corrected errors with refs *)
2 let ipow x n =   (* calculate x to the nth power *)
3   let p = ref 1 in
4   for i=1 to n do
5     p := !p * x; (* 1st error: get contents of p to multiply *)
6   done;
7   !p              (* 2nd error: return contents, not ref itself *)
8 ;;
9 (* File "ref_errors.ml", line 4, characters 9-10: *)
10   Error: This expression has type int ref
11   but an expression was expected of type int *)
12
13 let sum = (ipow 2 5) + (ipow 3 7);;
14 (* File "ref_errors.ml", line 12, characters 10-20: *)
15   Error: This expression has type int ref
16   but an expression was expected of type int *)
17
18 Printf.printf "sum is %d\n" sum;;
```
Exercise: Array Syntax, Predict Output

1 (* array_demo.ml : demonstrate array syntax *)
2 open Printf;;
3
4 (***** BLOCK 1 *****)
5 let arr = [|10; 20; 30; 40|] in (* immediate initialization *)
6 let len = Array.length arr in (* length calculation *)
7 printf "Length is %d\n" len;
8
9 (***** BLOCK 2 *****)
10 for i=0 to len-1 do
11  let eli = arr.(i) in (* access elements with arr.(i) *)
12  printf "El %d : %d\n" i eli;
13 done;
14
15 (***** BLOCK 3 *****)
16 printf "Doubling elements\n"; (* elements are mutable by default *)
17 for i=0 to len-1 do
18  arr.(i) <- arr.(i) * 2; (* assign with arr.(i) <- expr *)
19  printf "El %d : %d\n" i arr.(i);
20 done;
21
22 (***** BLOCK 4 *****)
23 let elem = "Monsier: répéter!" in
24 let big = Array.make 100 elem in (* 100 long array, filled with elem *)
25 for i=0 to (Array.length big)-1 do (* iterate over elements *)
26  printf "%s\n" big.(i); (* printing them *)
27 done;
28 ;;
Answers: Array Syntax, Predict Output

Output of array_demo.ml

> ocamlc array_demo.ml
> a.out |head -20

Length is 4  # BLOCK 1
El 0 : 10  # BLOCK 2
El 1 : 20
El 2 : 30
El 3 : 40
Doubling elements  # BLOCK 3
El 0 : 20
El 1 : 40
El 2 : 60
El 3 : 80
Monsier: répéter!  # BLOCK 4
Monsier: répéter!
Monsier: répéter!
Monsier: répéter!
Monsier: répéter!
... 100 times

Array Syntax Summary

(* immediate initialization *)
let arr = [|10; 20; 30; 40|] in

(* length calculation *)
let len = Array.length arr in

(* access elements with arr.(i) *)
let eli = arr.(i) in

(* assign with arr.(i) <- expr *)
arr.(i) <- x * 2;

(* Initialize and fill with elem *)
let big = Array.make 100 elem in
Arrays are bounds Checked

- Arrays are **fixed length** so growing them requires re-allocation
- Out of bounds access raises an exception

```ocaml
define arr = [|10; 20; 30; 40|];;
val arr : int array = [|10; 20; 30; 40|]
define arr.(3);
- : int = 40
define arr.(4);
Exception: Invalid_argument "index out of bounds".
define arr.(-5);
Exception: Invalid_argument "index out of bounds".
define arr.(7) <- 2;;
Exception: Invalid_argument "index out of bounds".
```

- Raised exceptions usually end a running program
- Can raise your own Failure exceptions if needed as in

```ocaml
define i < 2 then
  raise (Failure "Sainte merde!")
;;
Exception: Failure "Sainte merde!".
```

- Will explore exceptions in more detail later
Exercise: A Type Puzzle

Consider function `swap_0_1`

▶ What is it doing?
▶ What new syntax is present?
▶ What is the return type of the function?
▶ What is the type of parameter `arr`?

```ml
(* swap_0_1.ml : function with interesting type signature *)
let swap_0_1 arr =
if Array.length arr >= 2 then
begin
  let x = arr.(0) in
  let y = arr.(1) in
  arr.(0) <- y;
  arr.(1) <- x;
end;
;;
```
Consider function swap_0_1

- What is it doing? Swapping 0th and 1th elements of an array
- What new syntax is present? begin/end to include multiple side-effects statements in an if condition
- What is the return type of the function? unit as the last thing done is array assignment
- What is the type of parameter arr? 'a array???
  - any kind of array

```ocaml
doxygen
1 (* swap first two elems in an array *)
2 let swap_0_1 (arr : 'a array) : unit =
3 (* any array type return *)
4   if Array.length arr >= 2 then
5     begin (* begin a "block" within if *)
6       let x = arr.(0) in
7       let y = arr.(1) in
8       arr.(0) <- y; (* begin required as multiple *)
9       arr.(1) <- x; (* side-effects are performed *)
10      end;
11 ;; (* function returns unit *)
```
Polymorphism

polymorphism, (noun)

➢ The condition of occurring in several different forms.
➢ COMPUTING: a feature of a programming language that allows routines to use variables of different types at different times.

➢ A function is polymorphic if it works for a range of types
➢ The type signatures of these have ‘a or variants involved.

➢ Examples:

  ‘a -> int (* any type in, int out *)
  ‘a -> ‘a (* any type in, same type out *)
  ‘a -> ‘b (* any type in, any type out *)
  ‘a array -> int (* any type of array in, int out *)
  ‘a array -> ‘a (* any array in, element type out *)
  ‘a array -> ‘a array (* any array in, same type array out *)
  ‘a array -> ‘b array (* any array in, any array type out *)
  int -> ‘a -> ‘a (* int and any type in, out matches in type *)
  ‘a -> ‘a -> bool (* two args same kind in, bool out *)
  ‘a -> ‘b -> ‘a (* any two types in, first type out *)
  ‘a -> ‘b -> ‘c (* any two types in, any type out *)
Polymorphism Pervades OCaml

Polymorphism is everywhere in OCaml as evidenced by many built-in functions with polymorphic types

# (=);; (* comparisons *)
- : 'a -> 'a -> bool = <fun>
# ( > );;
- : 'a -> 'a -> bool = <fun>
# max;;
- : 'a -> 'a -> 'a = <fun> (* min/max *)
# min;;
- : 'a -> 'a -> 'a = <fun>

# ref;; (* ref operators *)
- : 'a -> 'a ref = <fun>
# (!);;
- : 'a ref -> 'a = <fun>
# (:=);;
- : 'a ref -> 'a -> unit = <fun>

# Array.make;; (* array functions *)
- : int -> 'a -> 'a array = <fun>
# Array.get;;
- : 'a array -> int -> 'a = <fun>
# Array.sub;;
- : 'a array -> int -> int -> 'a array = <fun>
Exercise: Writing Polymorphic Functions

Write the function `count_times elem arr`

- Counts how many times `elem` occurs in array `arr`
- Returns an `int`
- Ensure that operations performed are polymorphic
  - `=` operator checks equality, is polymorphic
  - Array access is polymorphic
- Should make function polymorphic with type
  `'a -> 'a array -> int`

REPL Demo of `count_times`

```ml
# #use "count_times.ml";;
val count_times : 'a -> 'a array -> int = <fun>
# count_times 4 [10; 2; 4; 1; 4; 11; 4; 7];;
- : int = 3
# count_times 11 [10; 2; 4; 1; 4; 11; 4; 7];;
- : int = 1
# count_times true [false; true; true; false; true];;
- : int = 3
# count_times "a" ["a"; "b"; "c"; "a"; "d"];;
- : int = 2
```
Answers: Writing Polymorphic Functions

1 (* count_times.ml : polymorphic counting function *)

2

3 (* count number of times elem appears in array arr *)

4 let count_times elem arr =

5 let count = ref 0 in (* ref to count *)

6 let len = Array.length arr in (* array length *)

7 for i=0 to len-1 do

8 if arr.(i) = elem then (* check for equal elem *)

9 count := !count + 1 (* update count if equal *)

10 (* incr count; *) (* increments an in ref *)

11 done;

12 !count (* deref count and return *)

13 ;;

General Guidelines for Polymorphic Functions

► Use only polymorphic operators like comparisons, assignments

► Polymorphism usually applicable to data structures like arrays, lists, tuples, trees, etc. that contain any kind of element

► Polymorphic funcs are more flexible, do it when you can

► In some cases, polymorphic functions are slower; explicitly typed versions can increase speed at the cost of flexibility