OCaml Revisited

Mooly Sagiv

Code from Ilya Sergey,
Real World OCaml

Factorial in OCaml

```
let rec fac n = if n = 0 then 1 else n * fac (n - 1)
```

```
val fac : int -> int = <fun>
```

```
let rec fac  n : int = if n = 0 then 1 else n * fac (n - 1)
```

```
let rec fac n = match n with
    | 0 -> 1
    | n -> n * fac(n - 1)
```

```
let rec fac  = function
    | 0 -> 1
    | n -> n * fac(n - 1)
```

```
let fac n =
    let rec ifac n acc =
        if n=0 then acc else ifac n-1, n * acc
    in ifac n, 1
```

Functions on Lists

```
let rec length l =
    match l with
   [] -> 0
  | hd :: tl  -> 1 + length tl
val length : 'a list -> int = <fun>
```

```
length [1; 2; 3] + length [“red”; “yellow”; “green”]
:- int = 6
```

```
length [“red”; “yellow”; 3]
```

Map Function on Lists

- Apply function to every element of list

```
let rec map f arg =
    match arg with

     [] -> []
    | hd :: tl -> f hd :: (map f tl)

val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

```
map (fun x -> x+1) [1;2;3]   ⇒   [2,3,4]
```

- Compare to Lisp

```
(define map
    (lambda (f  xs)
      (if   (eq? xs ())   ()
         (cons (f   (car xs))   (map f   (cdr xs)))
   )))
```

More Functions on Lists

- Append lists

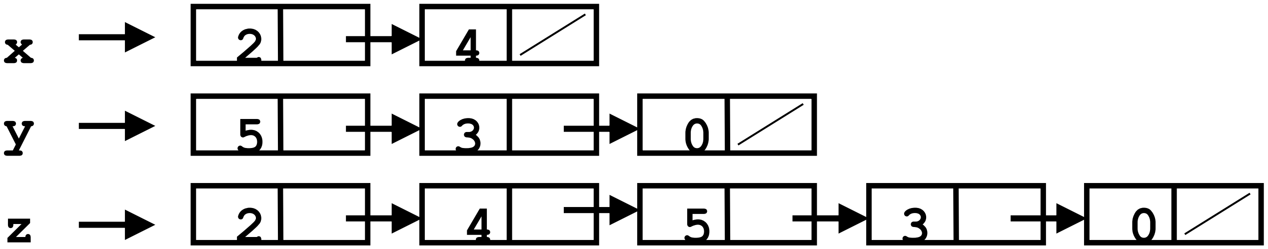
```
let rec append l1 l2 =
 match l1 with
 | []  -> l2
 | hd :: tl -> hd :: append (tl l2)
val append 'a list -> 'a list -> 'a list
```

```
let rec append l1 l2 =
 match l1 with
 | []  -> []
 | hd :: tl -> hd :: append (tl l2)
val append 'a list -> `b -> 'a list
```

List Example

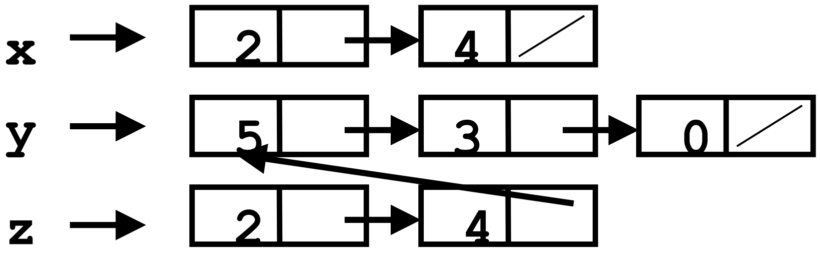
```
let rec append l1 l2 =
 match l1 with
  | []   -> l2
  | hd :: tl -> hd :: append tl l2
val append : 'a list -> 'a list -> 'a list = <fun>
```

```
let x = [2;4] //val x : int list = [2; 4]
let y = [5;3;0] //val y : int list = [5; 3; 0]
let z = append x y
  //val z : int list = [2; 4; 5; 3; 0]
```



(can't tell, but it's the second one)

or



More Functions on Lists

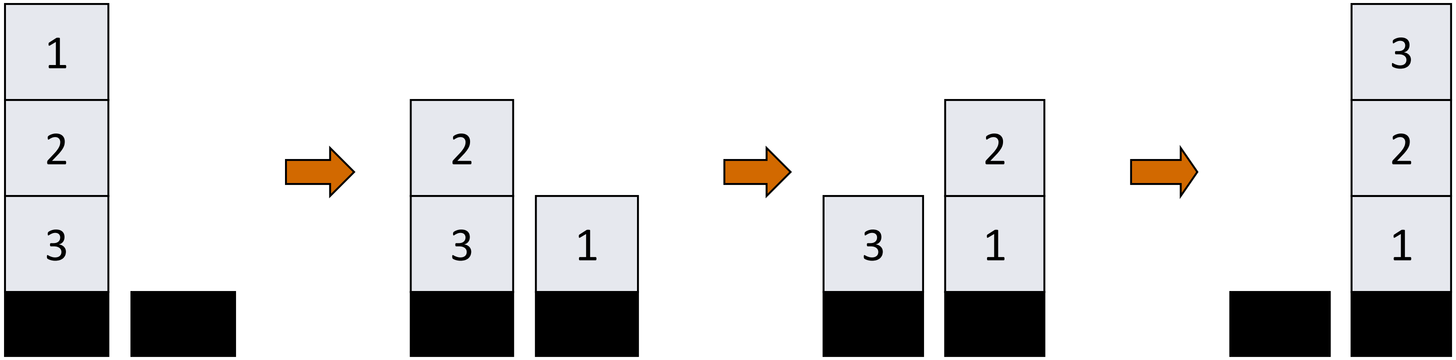
- Reverse a list

```
let rec reverse l =
   match l with
  | [] -> []
  | hd :: tl -> append (reverse tl)   [hd]
val reverse 'a list -> 'a list
```

- Questions
  - How efficient is reverse?
  - Can it be done with only one pass through list?

More Efficient Reverse

```
let rev list =
    let rec aux acc arg =
      match arg with
      [] -> acc
      | h::t -> aux (h::acc) t
    in
      aux [] list
val rev : 'a list -> 'a list = <fun>
```



Some Exercises

- Sum of the elements
- Drop an element
- Filter elements
- Insertion sort

Insertion Sort

```
let rec sort  ls = match ls with
    | [] -> []
    | x :: rest -> insert x (sort rest)
  and insert elem ls = match ls with
    | [] -> [elem]
    | x :: l -> if elem < x then elem :: x :: l
                  else x :: insert elem l
```

```
val sort : 'a list -> 'a list = <fun>
val insert : 'a -> 'a list -> 'a list = <fun>
```

```
sort [2; 1; 0];;
- : int list = [0; 1; 2]
```

```
sort ["yes"; "ok"; "sure"; "ya"; "yep"];;
- : string list = ["ok"; "sure"; "ya"; "yep"; "ye”]
```

Variant Records

- Provides a way to declare Algebraic data types

```
type expression = Number of  int | Plus of expression * expression
```

```
let rec eval_exp (e : expression)  =
  match e with
    Number(n) -> n
   | Plus (left, right) -> eval_exp(left) + eval_exp(right)
val eval_exp : expression -> int = <fun>
```

```
 eval_exp (Plus(Plus(Number(2), Number(3)), Number(5)))
:- int = 10
```

Algebraic Type for Naturals

```
type nat = zero | Succ of nat
```

```
let nat_to_int n = ….
```

```
let int_to_nat n = ….
```

List Algebraic Type

```
type ‘a list = empty | cons of ‘a *  list
```

Binary Search Trees

```
type 'a bst = Empty | Node of 'a bst * 'a * 'a bst
```

```
let rec find_max …
```

```
let rec insert  …
```

```
let rec delete  …
```

Insertion into a sorted tree

```
let rec insert x = function
   | Empty -> Node (Empty, x, Empty)
   | Node (l, y, r) ->
           if x = y then Node (l, y, r)
           else if x < y then Node (insert x l, y, r)
           else Node (l, y, insert x r)
```

```
val insert : 'a -> 'a bst -> 'a bst =
```

Modules & Side-effects

Benefits of Functional Programming

- No side-effects
- Referential Transparency
  - The value of expression e depends only on its arguments
- Conceptual
- Commutativity
- Easier to show that the code is correct
- Easier to generate efficient implementation

Side-Effects

- But sometimes side-effects are necessary
- The whole purpose of programming is to conduct side-effects
  - Input/Output
- Sometimes sharing is essential for functionality
- OCaml provides mechanisms to capture side-effects
  - Enable efficient handling of code with little side effects

OCaml Features for modularity

- modules organize identifiers (functions, values, etc.) into namespaces
- signatures
  - describe related modules
- abstract types
  - control what is visible outside a namespace

Stack with Abstract Data Types

```
module type STACK = sig
  type t
  val empty : t
  val is_empty : t -> bool
  val push : int -> t -> t
  val pop : t -> int * t
end
module Stack : STACK = struct
  type t = int list
  let empty = [ ]
  let is_empty s = s = [ ]
  let push x s = x :: s
  let pop s = match s with
      [ ] -> failwith "Empty”
      | x::xs -> (x,xs)
end
```

Input/Output

print_string: String -> Unit

```
let x = 3 in
  let () = print_string ("Value of x is " ^ (string_of_int x)) in
  x + 1
value of x is 3- : int = 4
```

e ::= ... | (e_1; ... ; e_n)

```
let x = 3 in
  (print_string ("Value of x is " ^ (string_of_int x));
   x + 1)
```

Iterative loops are supported too

Refs and Arrays

- Two built-in data-structures for implementing shared objects

```
module type REF =
  sig
    type 'a ref
    (* ref(x) creates a new ref containing x *)
    val ref : 'a -> 'a ref
    (* !x is the contents of the ref cell x *)
    val (!) : 'a ref -> ‘a
    (* Effects: x := y updates the contents of x
     * so it contains y. *)
    val (:=) : 'a ref -> 'a -> unit
  end
```

Simple Ref Examples

```
let x : int ref = ref 3
  in
    let y : int = !x
     in
        (x := !x + 1);
        y + !x
- : int = 7
```

More Examples of Imperative Programming

- Create cell and change contents

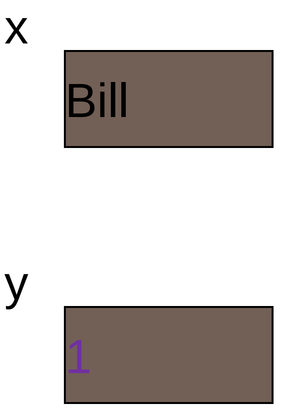
```
let x = ref "Bob";
x := "Bill";
```

- Create cell and increment

```
let y = ref 0;
y := !y + 1;
```

- While loop

```
let i = ref 0;
while !i < 10 do i := !i +1;
 i;
```



Imperative Loops

- Combine normal control flow structures and functional programming

```
for i = 1 to 100  do
 Printf.printf (“%d”, fact i)
done
```

Imperative Factorial

```
let fact n =
    let result = ref 1 in
    for i = 2 to n do
      result := i * !result
    done;
    !result;;
val fact : int -> int = <fun>
```

Fibonacci Numbers

- A sequence
  - $fib_0=1$
  - $fib_1 =1$
  - $fib_n = fib_{n-2}+fib_{n-1}$

1 1 2 3 5

Fibonacci in OCaml

```
let rec fib n =
  match n with
   | 0 -> 1
   | 1 -> 1
   | n -> n * fib(n - 1)
```

```
let rec fibonacci n =
  if n < 3 then
    1
  else
    fibonacci (n-1) + fibonacci (n-2)
```

```
let () =
  for n = 1 to 16 do
    Printf.printf "%d, " (fibonacci n)
  done;
  print_endline "..."
```

Fibonacci in OCaml

```
let rec fib n =
  match n with
   | 0 -> 1
   | 1 -> 1
   | n -> n * fib(n - 1)
```

How efficient is fib(n)?

time (fun () -> fib 20);

Time: 0.379086ms - : int = 10946

time (fun () -> fib 40);

Time: Time: 4.61983s - : int = 165580141

Memoized Fibonacci

```
let memo_fib n =
  if n <= 1 then 1
  else begin
    let fib = ref  1 in
    let fib_prev = ref 1 in
    for i = 2 to n do
       let tmp = !fib_prev in
       fib_prev := !fib;
       fib := tmp + !fib;
       done;
    !fib
  end
```

Testing the results

```
let test_fib n =
  for i = 0 to n do
    assert (memo_fib n = fib n)
  done;
  true
```

```
time fib 38;;
Execution elapsed time: 1.433646 sec
- : int = 63245986
time memo_fib 38;;
Execution elapsed time: 0.000008 sec
- : int = 63245986
```

Associative Maps

```
type ‘k ‘v assocMap = emptyMap | cons of ‘k * ‘v * assocMap
```

```
let rec access m index =
        match m with
         |  emptyMap          -> raise
              (FailWith “Element not found“^ (string_of_int index))
         | cons(k, v, rest)  ->
              if k = index then v else access rest index
```

```
let rec set m index newValue =
        match m with
         |  emptyMap          -> cons(index, newValue, emptyMap)
         |  cons(k, v, rest)  ->  if k = index then
                                     cons(index, newValue, rest)
                                          else
                                     cons(k, v, set(rest, index, newValue))
```

Arrays

- A ML module provides efficient constant time array access
  - <array_expr>.(<index_expr>)
  - <array_expr>.(<index_expr>) <- <value_expr>

Array Example

```
let add_polynom p1 p2 =
    let n1 = Array.length p1
    and n2 = Array.length p2 in
    let result = Array.make (max n1 n2) 0 in
    for i = 0 to n1 - 1 do result.(i) <- p1.(i) done;
    for i = 0 to n2 - 1 do result.(i) <- result.(i) + p2.(i) done;
    result
```

```
val add_polynom : int array -> int array -> int array = <fun>
```

```
add_polynom [| 1; 2 |] [| 1; 2; 3 |];
```

```
- : int array = [|2; 4; 3|]
```

In situ reverse

```
let rev_inplace ar =
  let i = ref 0 in
  let j = ref (Array.length ar - 1) in
  (* terminate when the upper and lower indices meet *)
  while !i < !j do
    (* swap the two elements *)
    let tmp = ar.(!i) in
    ar.(!i) <- ar.(!j);
    ar.(!j) <- tmp;
    (* bump the indices *)
    Int.incr i;
    Int.decr j
  done
```

```
let nums = [|1;2;3;4;5|];;
```

```
rev_inplace nums;
```

OCaml Advanced Modularity Features

- Functors and Signatures
- Functions from Modules to Modules
- Permit
  - Dependency injection
  - Swap implementations
  - Advanced testing

Summary Modularity

- ML provides flexible mechanisms for modularity
- Guarantees type safety

Summary References

- Provide an escape for imperative programming
- But insures type safety
  - No dangling references
  - No (double) free
  - No null dereferences
- Relies on automatic memory management

Functional Programming Languages

PL	types	evaluation	Side-effect
scheme	Weakly typed	Eager	yes
OCaml OCAOCaml F#	Polymorphic strongly typed	Eager	References
Haskell	Polymorphic strongly typed	Lazy	None

Recommended ML Textbooks

- L. C. PAULSON: ML for the Working Programmer
- J. Ullman: Elements of ML Programming
- R. Harper: Programming in Standard ML

Recommended OCaml Textbooks

- Xavier Leroy: The OCaml system release 4.02
  - Part I: Introduction
- Jason Hickey: Introduction to Objective Caml
- Yaron Minsky, Anil Madhavapeddy, Jason Hickey: Real World OCaml

Summary

- Functional programs provide concise coding
- Compiled code compares with C code
- Successfully used in some commercial applications
  - F#, ERLANG, Jane Street, Scala. Kotlin
- Ideas used in imperative programs
- Good conceptual tool