Lecture 11
First Order Functional Programming with Mutable Values

prof. Marco Bellia, Dip. Informatica, Università di Pisa

March 21, 2014

A First Look to Functional Programming in Caml

- Functional Languages: The main Features
- First Order Programming in Caml
- Mutable Values in Functional Programming

Functional Languages: The main Features

- Referential Transparency (Pure Functional)
- First-Class Function Values and Higher Order Functions (not here)
- Extensive Polymorphism
- List: Types and Operators
- Functions may return Structured Values
- Structured Values are Fully Expressible Values
- Garbage Collection for Heap re-allocation

Preliminaries

- The support: http://caml.inria.fr/
- Documentation and users manual: The Objective Caml system (same site)
- Once installed: "read-eval-print" interpretation cycle

Example

```
bellia:~ marcobellia$ ocaml
      Objective Caml version 3.11.1
# (3+4);;
- : int = 7
# ((3+4));;          -- significance of parentheses
- : int = 7
# let x = 5;;        -- binding in the main environment
val x : int = 5
# if x>3 then x else 3;;      -- type checking
- : int = 5
```

Module, Polymorphism an List

- Modularization in files
- Extensive Polymorphism
- List: Types and Operators

Example

```
# #use "List.ml";;                    -- Modularization and loading
val length_aux : int -> 'a list -> int = <fun>
val length : 'a list -> int = <fun>     -- Polymorphism 'a list -> int for all 'a
...
# [2;3+5;x];;
- : int list = [2; 8; 5]                -- 'a list becomes int list
# [3.14;"string"];;                     -- Polymorphism: wrong use of Poly.
Error: This expression has type string but an expression was expected of type float
# [3.14,"string"];;                   -- Pair Constructor
- : (float * string) list = [(3.14, "string")]
```

Functions may return Structured Values

- Referential Transparency is incompatible with Side Effects
- Hence, Pure Functional Programming forbids Side Effects
- Hence, Functional Programming cannot use Mutable Values
- Hence, **Update one component of a structured value**
  must be rephrased in
  Compute a new structured value that differs for the component to update

Example

```
# let sum = fun x y -> x+y;;    -- binding with a lambda abstraction
val sum : int -> int -> int = <fun>
# sum 2 3;;
- : int = 5
# sum 2;;                                -- Curryfication: this value is a function
- : int -> int = <fun>
# let sumP = fun (x,y) -> x+y;;          -- Uncurryfied definition of sum
val sumP : int * int -> int = <fun>
# let rec replace = fun n list k ->  if n=1 then k::(tl list) else (hd list)::(replace (n-1)(tl list)
k);;
val replace : int -> 'a list -> 'a -> 'a list = <fun>     -- "replacing" n-th element of list with k
# let a = [3;5;7;13];;
val a : int list = [3; 5; 7; 13]
# replace 3 a 8;;                                  -- replace computes a new structure value
- : int list = [3; 5; 8; 13]
# a;;
- : int list = [3; 5; 7; 13]
```

Block and Nested Definitions

- Static Scope
- Correlated and Uncorrelated Simultaneous Definitions

Example

```
# let x = 5 in                                  -- inline blocks
  let x = 3 and y = x in x+y;;                  -- simultaneous uncorrelated
- : int = 8
# let x=3 and y=x in x+y;;                      -- uncorrelated
Error: Unbound value x
let rec x = 4 and y = [x;7;9] in y;;            -- parallel or simultaneous correlated
- : int list = [4; 7; 9]
# let rec  y = [x;7;9] and x = 2 in y;;         -- parallel: ordering is unessential
- : int list = [2; 7; 9]
# let rec x=3 and y=x in x+y;;                  -- parallel or simultaneous correlated
Error: This kind of expression is not allowed as right-hand side of `let rec'
# let rec x = 3 and y = [x;x+2] in y;;                  --  parallel or simultaneous correlated: Why?
Error: This kind of expression is not allowed as right-hand side of `let rec'
# let rec x = 3 and y = fun unit -> [x;x+2] in y();;    -- parallel or simultaneous correlated: Why?
- : int list = [3; 5]                          -- function blocks: Scope is static
# let w = 5;;
val w : int = 5
# let g = fun x -> x+w;;                                -- Scope is static
val g : int -> int = <fun>
# let h = fun x -> let w = 12 in w + (g x);;            -- Scope is static
val h : int -> int = <fun>
# h 0;;
- : int = 17                                           -- What value should be printed in case of dynamic scope
# let rec natAdd(n,m) = if (n=0) then m else 1+natAdd(n-1,m);;  -- recursive function
val natAdd : int * int -> int = <fun>                           -- undefined on negative integers
```

Structured Values are Fully Expressible Values

- Fully Expressible = All Values (but functions), have a presentation
- Hence all Value Constructors have a concrete presentation
- Pattern Matching in by Case Definitions

Example

```
# let rec rev = fun l -> match l with      -- Construct: match E with Case_Pairs+
        |[] -> []                          -- by case definition
        |x::lR -> append (rev lR) [x]
        ;;
val rev : 'a list -> 'a list = <fun>
# let rec filter p xs = match xs with
    [] -> []
    |y::ys when (p y) -> y::(filter p ys)   -- when E: Restriction in by case definition
    |otherwise -> filter p (tl xs)
    ;;
val filter : ('a -> bool) -> 'a list -> 'a list = <fun>
# type intList = Null | Cons of int * intList;;     -- constructed types (algebraic types, concrete types)
type intList = Null | Cons of int * intList         -- recursive constructed types
# let rec append = fun x y -> match x with          -- append on constructed intList
  Null -> y
  |Cons(n,rest) -> Cons(n,(append rest y));;
val append : intList -> intList -> intList = <fun>
# append (Cons(2,Cons(3,Null))) (Cons(4,Null));;    -- append on constructed intList
- : intList = Cons (2, Cons (3, Cons (4, Null)))
# type 'a alist = Nil | Cons of 'a * 'a alist;;     -- polymorphic constructed types (algebraic types, concrete types)
type 'a alist = Nil | Cons of 'a * 'a alist
```

Undefined Values, Exceptions: raised and trapped

- Undefined Values = partial functions
- Exception for controlling the executions

Example

```
# type 'a alist = Nil | Cons of 'a * 'a alist;;     -- polymorphic constructed types (algebraic types, concrete types)
type 'a alist = Nil | Cons of 'a * 'a alist
# let intUndef = let rec f x = if ((f x) = x) then 1 else 0 in (f 0);;   -- undefined of int
Stack overflow during evaluation (looping recursion?).  -- nonterminating computation of type int
# let rec aFun x = if ((aFun x) = x) then x else x;;    -- a function for aUndef
val aFun : 'a -> 'a = <fun>                             -- what is the type and the value of: aFun Nil
# let tail = fun x -> match x with                      -- tail returns an undefined value when the list is empty
  Nil -> aFun Nil                                       -- aFun Nil provides a nonterminating computation
  |Cons(u,us) -> us;;
val tail : 'a alist -> 'a alist = <fun>
# exception EmptyList;;                          -- use of exception
exception EmptyList                              -- is a special class of values
# let tail = fun x -> match x with               -- tail returns an exception when the list is empty
  Nil -> raise EmptyList                         -- raising an execption
  |Cons(u,us) -> us;;
val tail : 'a alist -> 'a alist = <fun>
# tail Nil;;
Exception: EmptyList.
# let tailTotal x = try tail x with EmptyList -> Nil;;  -- tailTotal: Exception for the empty list is trapped
val tail : 'a alist -> 'a alist = <fun>
# tailTotal Nil;;
- : 'a alist = Nil
```

Mutable Values in Ocaml

- Any value, included functions, can be made a Mutable Value

Example

```
# let x = ref 10;;
val x : int ref = {contents = 10}        -- mutable int with associated 10
# x;;
- : int ref = {contents = 10}            -- mutable value bound to x
# !x;;
- : int = 10                             -- associated value of the mutable bound to x
# let x = ref [];;
val x : '_a list ref = {contents = []}   -- mutable polimorphic empty list
# x:=3::!x;;
- : unit = ()
# !x;;
- : int list = [3]                        -- modifying mutable polymorphic lists
# let f = ref (fun x -> x);;              -- mutable polymorphic function
val f : ('_a -> '_a) ref = {contents = <fun>}
# !f(5);;                                 -- using the associated function
- : int = 5
# f:= fun x y -> filter x y;;             -- assigning a value of incorrect type
Error: This function expects too many arguments, it should have type
int -> int
# f:= fun x -> x+1;;                      -- assigning a value of correct type
- : unit = ()
# !f(5);;
- : int = 6
```

Exercises

1. Complete with the suitable definitions in order to run the following Ocaml codes

Example

```
let x = ref 0 in
 let pippo xr =
      function n -> xr := !xr + n in
  let pippo1 = pippo(x) in
   pippo1(3);
   print(!x);
   (let x = ref 0 in
      pippo1(3);
      print(!x));
   print(!x);
```

```
let x = ref 0 in
 let pippo xr =
      function n -> xr := !xr + n in
  pippo(x)(3);
  print(!x);
  (let x = ref 0 in
     pippo(x)(3);
     print(!x));
  print(!x);
```

2. Give definitions for the domain Env: Values and Operations
3. Give definitions for the domain Store: Values and Operations