Introduction to ML

Mooly Sagiv

Cornell CS 3110 Data Structures and Functional Programming
The ML Programming Language

• General purpose programming language designed by Robin Milner in 1970
  – Meta Language for verification
• Can be viewed as typed λ calculus with type inference
• Impure Functional Programming Language
  – Eager call by value evaluation
• Static strongly typed (like Java unlike C)
  – Protect its abstraction via type checking and runtime checking
• Polymorphic Type Inference
• Dialects: OCaml, Standard ML, F#
C is not Type Safe

```c
int j;
union { int i, int * p } x;
x.i = 17;
j = *(x.p);

int i, *p;
i = 17
p = (int *) i;
```
Factorial in ML

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

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

int fac(int n) {
    if (n == 0) return 1; else return n * fac (n - 1); 
}
Factorial in ML

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 = 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
Why Study ML?

• Functional programming will make you think differently about programming
  – Mainstream languages are all about state
  – Functional programming is all about values

• ML is “cutting edge”
  – Polymorphic Type inference
  – References
  – Module system

• Practical (small) Programming Language
• Embedded in Java 8, Scala/Kotlin
• New ideas can help make you a better programmer, in any language
Plan

- Basic Programming in ML
- ML Modules & References
- Closure
- Type Inference for ML
Simple Types

- **Booleans**
  
  ```
  true -: bool = true
  false -: bool = false
  if ... then ... else ... types must match
  ```

- **Integers**
  
  ```
  0, 1, 2, ... -: int =0, 1, ...
  +, *, -: int * int -> int
  ```

- **Strings**
  
  “I am a string” -: string = “I am a string”

- **Floats**
  
  ```
  1.0, 2., 3.14159, ... :- float = 1, 2, 3.14159
  ```
Scope Rules

- ML enforces static nesting on identifiers
  - To be explained later
- \texttt{let } x = e_1 \texttt{ in } e_2 \equiv (\lambda x. e_2) \ e_1
- \texttt{let } x: T = e_1 \texttt{ in } e_2 \equiv (\lambda x: T. e_2) \ e_1
Tuples

4, 5, “abc” :- (int*int*string)=(4, 5, “abc”)

let max1 (r1, r2) : float = 
  if r1 < r2 then r2 else r1
val max1: float * float -> float = fun

let args = (3.5, 4.5)
val args: float * float = (3.5, 4.5)

max1 args
  :-  float = 4.5
Pattern-Matching Tuples

let \( y(x_1 : t_1, x_2 : t_2, \ldots, x_n : t_n) = e \)

let max1 (pair : float * float) : float =
  let (r1, r2) = pair in
  if r1 < r2 then r2 else r1
val max1 : float * float -> float = fun

let minmax (a, b) : float * float =
  if a < b then (a, b) else (b, a)
val minmax : float * float -> float * float = fun

let (mn, mx) = minmax (2.0, 1.0)
val mn float 1
val mx float 2

The compiler guarantees the absence of runtime errors
User-Defined Types

define a type for days of the week:

```plaintext
type day = Sun | Mon | Tue | Wed | Thu | Fri | Sat
```

and a function `int_to_day` to convert an integer to a day:

```plaintext
let int_to_day (i : int) : day =
  match i mod 7 with
    | 0 -> Sun
    | 1 -> Mon
    | 2 -> Tue
    | 3 -> Wed
    | 4 -> Thu
    | 5 -> Fri
    | _  -> Sat
```
User-Defined Types C

```c
enum day { Sun, Mon, Tue, Wed, Thu, Fri, Sat };

day int_to_day (int i) {
    switch i % 7 {
    case 0: return Sun ;
    case 1: return Mon ;
    case 2: return Tue ;
    case 3: return Wed ;
    case 4: return Thu ;
    case 5: return Fri ;
    default: return Sat ;
    }
}
```
Records

```haskell
type person = {first:string; last:string; age:int}

{first="John"; last="Amstrong"; age=77}
:: person ={first="John; last="Amstrong"; age=77}

{first="John"; last="Amstrong"; age=77}.age
:: int = 77

let ja = {first="John"; last="Amstrong"; age=77}
val ja : person = {first="John"; last="Amstrong"; age=77}

let {first=first; last=last} = ja
val first:string="John"
val last:string ="Amstrong"
```
C structs

```c
struct person { char * first; char * last; int age };

struct person ja = { "John", "Amstrong", 77 };

printf("%d", ja.age);

fn = ja.first; ln = ja.last; ag = ja.age

ja.age = 78;
```
Variant Records

- Provides a way to declare Algebraic data types

```plaintext
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>
```

```plaintext
let rec eval_exp = function
  | Number(n) -> n
  | Plus (left, right) -> eval_exp(left) + eval_exp(right)

val eval_exp : expression -> int = <fun>
```

```plaintext
eval_exp (Plus(Plus(Number(2), Number(3)), Number(5)))
:- int = 10
```
Variant Records in C

```c
struct exp {
    enum {Number, Binop} etype ; /* Select between cases */
    union {
        struct number { int : num; }
        struct plus { struct exp *left, *right; }
    }
}

int eval_exp (struct exp e) {
    switch e.etype {
    case Number: return e.number.num ;
    case Binop : return eval_exp(e.plus.left) + eval_exp(e.plus.right);
    }
}
```
Wrong Variant Records in C

```c
struct exp {
    enum {Number, Binop} etype; /* Select between cases */
    union {
        struct number { int num; }
        struct plus { struct exp *left, *right; }
    }
}

int eval_exp (struct exp e) {
    switch expression.etype {
        case Binop: return e.number.num ;
        case Number : return eval_exp(e.plus.left) + eval_exp(e.plus.right);
    }
}
```
Define an Algebraic Type for Naturals
Int List Algebraic Type
Binary Search Tree

type tree = Leaf of int | Node of tree * tree * int

let rec search t key =
  match t with
  Leaf(n) -> key = n
| Node (left, right, n) -> if key = n
    then true
    else if key < n
      then search left key
      else search right key

Scope

• Local nested scopes
• Let constructs introduce a scope

```plaintext
let f x = e1 in e2

let x = 2
and y = 3
in x + y

let rec even x = x = 0 || odd (x-1)
    and odd x = not (x = 0 || not (even (x-1)))
in
odd 3110
```
Polymorphism

• Functions can have many types
  – \( \lambda x. \ x \)
  – \( \lambda f. \ \lambda g. \ \lambda x. \ f (g \ x) \)
Polymorphism

- A Polymorphic expression may have many types
- There is a “most general type”
- The compiler infers types automatically
- Programmers can restrict the types
- Pros:
  - Code reuse
  - Guarantee consistency
- Cons:
  - Compilation-time
  - Some limits on programming

``` OCaml
let max1 (r1, r2) = 
  if r1 < r2 then r2 else r1 
val max1: 'a * 'a -> 'a = fun
```

``` OCaml
max1 (5, 7) 
: int = 7
```

``` OCaml
max1 (5, 7.5)
```
Polymorphic Lists

- : 'a list = []

- : int list = [2; 7; 8]

- : int list = [2; 7; 8]

((2, 7); (4, 9); 5]

Error: This expression has type int but an expression was expected of type int * int
Functions on Lists

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

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

```ocaml
length ["red"; "yellow"; 3]
```
Higher Order Functions

• Functions are first class objects
  – Passed as parameters
  – Returned as results
• Practical examples
  – Google map/reduce
Map Function on Lists

• Apply function to every element of list

```ocaml
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

```lisp
(define map    
  (lambda (f xs)    
    (if (eq? xs ()) ()    
      (cons (f (car xs)) (map f (cdr xs))))))
```
More Functions on Lists

• Append lists

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

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

• Reverse a list

```ocaml
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?
let rev list =
  let rec aux acc = function
    | [] -> acc
    | h::t -> aux (h::acc) t
  in
  aux [] list
val rev : 'a list -> 'a list = <fun>
Currying

```ocaml
let plus (x, y) = x + y
val plus : int * int -> int = fun

let plus (z : int * int) = match z with (x, y) -> x + y
val plus : int * int -> int = fun

let plus = fun (z : int * int) -> match z with (x, y) -> x + y
val plus : int * int -> int = fun

let plus x y = x + y
val plus : int -> int -> int

let p1 = plus 5
val p1 : int -> int = fun

let p2 = p1 7
val p2 : int = 12
```
## Functional Programming Languages

<table>
<thead>
<tr>
<th>PL</th>
<th>types</th>
<th>evaluation</th>
<th>Side-effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>scheme</td>
<td>Weakly typed</td>
<td>Eager</td>
<td>yes</td>
</tr>
<tr>
<td>ML OCAML F#</td>
<td>Polymorphic strongly typed</td>
<td>Eager</td>
<td>References</td>
</tr>
<tr>
<td>Haskell</td>
<td>Polymorphic strongly typed</td>
<td>Lazy</td>
<td>None</td>
</tr>
</tbody>
</table>
Things to Notice

• Pure functions are easy to test

prop_RevRev l = reverse(reverse l) == l

• In an imperative or OO language, you have to
  – set up the state of the object and the external state it reads or writes
  – make the call
  – inspect the state of the object and the external state
  – perhaps copy part of the object or global state, so that you can use it in the post condition
Things to Notice

Types are everywhere.

reverse:: [w] -> [w]

• Usual static-typing panegyric omitted...
• In ML, types express high-level design, in the same way that UML diagrams do, with the advantage that the type signatures are machine-checked
• Types are (almost always) optional: type inference fills them in if you leave them out
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
• Ideas used in imperative programs
• Good conceptual tool
• Less popular than imperative programs