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
• 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);
```

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

```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 = 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
• New ideas can help make you a better programmer, in any language
Plan

- Basic Programming in ML
- Type Inference for ML
- ML Modules & References
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
  ```
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

let y(x1: t1, x2: t2, ..., xn: tn) = e
Pattern-Matching Tuples

**Code Snippet 1:**

```
let x₁ : t₁, x₂ : t₂, ..., xₙ : tₙ = e
```

**Code Snippet 2:**

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

**Code Snippet 3:**

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

**Code Snippet 4:**

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

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
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"
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) : int =
  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
```
Variant Records in C

```c
struct exp {
    int tag ; /* Select between cases */
    union {
        struct number { int : number; }  
        struct plus { struct exp *left, *right; }  
    }
}
```
Scope

• Local nested scopes
• Let constructs introduce a scope

\[
\text{let } f \ x = e_1 \ \text{in} \ e_2
\]

\[
\text{let } x = 2 \\
\text{and } y = 3 \\
\text{in} \ x + y
\]

\[
\text{let rec } \text{even } x \ = \ x = 0 \ || \ \text{odd } (x-1) \\
\text{and } \text{odd } x \ = \ \text{not } (x = 0 \ || \ \text{not } (\text{even } (x-1))) \\
\text{in} \\
\text{odd } 3110
\]
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:
  - Compile-time
  - Some limits on programming

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

max1 (5, 7)
: - int = 7

max1 (5, 7.5)
```
Polymorphic Lists

[]
- : 'a list = []

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

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

```
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]
```
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

```ml
let rec map f arg = function
  [] -> []
  | hd :: tl -> f hd :: (map f tl)

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

`map (fun x -> x+1) [1;2;3] \rightarrow [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

\[
\text{let rec append \( l_1 \ l_2 \) =} \\
\quad \text{match \( l_1 \) with} \\
\quad | \[] \rightarrow l_2 \\
\quad | \text{hd :: tl} \rightarrow \text{hd :: append (tl} l_2) \\
\]

• Reverse a list

\[
\text{let rec reverse \( l \) = function} \\
\quad | \[] \rightarrow [] \\
\quad | \text{hd :: tl} \rightarrow \text{append (reverse tl} \ [\text{hd}]) \\
\]

• Questions
  
  – How efficient is reverse?
  
  – Can it be done with only one pass through list?
More Efficient Reverse

```ocaml
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

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

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
  
  \[ \text{prop\_RevRev}\ l = \text{reverse}(\text{reverse}\ l) == l \]

• 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.

• 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