CSE 130: Spring 2012
Programming Languages

Lecture 2: A Crash Course in ML

Ranjit Jhala
UC San Diego

News

On webpage:
• Suggested HW #1
• PA #1 (due next Fri 4/13)

Please post questions to Piazza

Today: A crash course in ML contd...

Recap: ML’s Holy Trinity

1. Programmer enters expression
2. ML checks if expression is “well-typed”
   • Using a precise set of rules, ML tries to find a unique type for the expression meaningful type for the expr
3. ML evaluates expression to compute value
   • Of the same “type” found in step 2

Complex types: Lists

<table>
<thead>
<tr>
<th>Expressions (Syntax)</th>
<th>Values (Semantics)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exec-time “Dynamic”</td>
<td>Types</td>
</tr>
<tr>
<td>Compile-time “Static”</td>
<td></td>
</tr>
</tbody>
</table>

1. []; 1:2:3; [1+1;2+2;3+3;4+4]; [“a”;“b”; “cd”]; [(1,”ab”);(7,”c”)]; [[1];[2;3];[4;5;6]];
2. [ ]; [1;2;3]; [2;4;6;8]; [“a”;“b”; “cd”]; [“a”;“b”; “cd”]; [(int*string) list]
3. [1]; [1;2;3]; [2;4;6;8]; [“a”;“b”; “cd”]; [(int*string) list]
Complex types: Lists

All elements must have same type

```
[1; “pq”];
```

List operator “Cons” ::

```
1::[2;3];
```
```
[1;2;3]
```
```
1::[“b”; “cd”];
```
```
[“a”;“b”;“c”]
```
```
a::[“b”; “c”];
```
```
[“a”;“b”;“c”]
```

Can only “cons” element to a list of same type

```
1::[“b”; “cd”];
```

Lists: Construct

Nil operator

```
[]
```
```
[] : ’a list
```
```
[] => []
```

Cons operator

```
1::[2;3]
```
```
[1;2;3]
```
```
e1:T e2: T list
```
```
e1=>v1 e2=> v2
```
```
e1::e2 : T list
e1::e2 => v1::v2
```

List operator “Append” @

```
[1;2]@[3;4;5];
```
```
[1;2;3;4;5]
```
```
[“a”]@[“b”];
```
```
[“a”;“b”]
```
```
[1]@[1];
```
```
[1]
```
```
1 @ [2;3];
```
```
[1] @ [“a”;“b”];
```

Can only append two lists of the same type
Complex types: Lists

List operator “head” \( \text{hd} \)

\[
\text{hd} \ [1;2]; \\
\text{hd} \ (\text{"a"}@[\text{"b"}]);
\]

Only take the head a nonempty list \( \text{hd} \ [ ]; \)

Complex types: Lists

List operator “tail” \( \text{tl} \)

\[
\text{tl} \ [1;2;3]; \\
\text{tl} \ (\text{"a"}@[\text{"b"}]);
\]

Only take the tail of nonempty list \( \text{tl} \ [ ]; \)

Lists: Deconstruct

Head

\[
e : T \text{ list} \\
\text{hd} \ e : T \\
\text{hd} \ e => v1::v2 \\
\]

Tail

\[
e : T \text{ list} \\
\text{tl} \ e : T \text{ list} \\
\text{tl} \ e => v2 \\
\]

Recap: Tuples vs. Lists?

What’s the difference?

- **Tuples:**
  - Different types, but fixed number:
    - Pair = 2 elts
    - Triple = 3 elts
    - (3, “abcd”) (int * string)
    - (3, “abcd”, (3.5, 4.2)) (int * string * (float * float))
  - Lists:
    - Same type, unbounded number:
      - [3;4;5;6;7] int list
So far, a fancy calculator... 

... what do we need next?

---

Branches

---

If-then-else expressions

• Then-subexp, Else-subexp must have same type!
  - Equals type of resulting expression

\[
\text{if 1>2 then [1,2] else []} \]
\[
\text{if 1<2 then [] else ["a"]}
\]

\( (\text{if 1>2 then [1,2] else []}) = (\text{if 1<2 then [] else ["a"]}) \)

---

If-then-else expressions

• then-subexp, else-subexp must have same type!
  - ...which is the type of resulting expression

\[
\text{if (1 < 2) then [1;2] else 5}
\]
\[
\text{if false then [1;2] else 5}
\]
Variables and bindings

Later declared expressions can use `x`  
- Most recent "bound" value used for evaluation

```
# let x = 2+2;;
val x : int = 4
# let y = x * x * x;;
val y : int = 64
# let z = [x;y;x+y];;
val z : int list = [4;64;68]
#
```

Undeclared variables  
(i.e. without a value binding)  
are not accepted!

```
# let p = a + 1;
Characters 8-9:
  let p = a + 1 ;;
^  
Unbound value a
```

Catches many bugs due to typos
Local bindings
... for expressions using “temporary” variables

```
let
  tempVar = x + 2 * y
in
  tempVar * tempVar
;;
```

- `tempVar` is bound only inside expr body from in ...
- Not visible (“not in scope”) outside

Binding by Pattern-Matching
Simultaneously bind several variables

```
# let (x,y,z) = (2+3,"a"^"b", 1::[2]);;
val x : int = 5
val y : string = "ab"
val z : int list = [1;2]
```

Binding by Pattern-Matching
But what of:

```
# let h::t = [1;2;3];;
Warning P: this pattern-matching not exhaustive.
val h : int = 1
val t : int list = [2;3]
```
Why is it whining?

```
# let h::t = []; Exception: Match_failure
# let XS = [1;2;3];
val xs = [1;2;3]: list - val h::t = xs;
Warning: Binding not exhaustive
val h = 1 : int
val t = [2;3] : int
```

In general `XS` may be empty (match failure!)

Another useful early warning

Functions
Next class: functions, but remember ...

Expression → Value → Type

Everything is an expression
Everything has a value
Everything has a type

A function is a value!

Complex types: Functions!

Parameter (formal) → Body Expr

fun x -> x+1;;

Expression → Value

int -> int

fun (x,y) -> x<y;

(int * int) -> bool

Functions only have ONE parameter ?!

A Problem

Functions only have ONE parameter ?!

A Solution: Simultaneous Binding

How a call ("application") is evaluated:
1. Evaluate argument
2. Bind formal to arg value
3. Evaluate "Body expr"

How a call ("application") is evaluated:
1. Evaluate argument
2. Bind formal to arg value
3. Evaluate "Body expr"
## Another Solution ("Currying")

<table>
<thead>
<tr>
<th>Parameter (formal)</th>
<th>Body</th>
<th>Expr</th>
</tr>
</thead>
<tbody>
<tr>
<td>fun x -&gt;</td>
<td>fun y -&gt;</td>
<td>x &lt; y;</td>
</tr>
<tr>
<td></td>
<td>fn</td>
<td></td>
</tr>
</tbody>
</table>

Whoa! A function can **return a function**

```plaintext
# let lt = fun x -> fun y -> x < y;
val lt : int -> int -> bool = fn
# let is5Lt = lt 5;
val is5Lt : int -> bool = fn;
# is5Lt 10;
val it : bool = true;
# is5Lt 2;
val it : bool = false;
```

## And how about...

<table>
<thead>
<tr>
<th>Parameter (formal)</th>
<th>Body</th>
<th>Expr</th>
</tr>
</thead>
<tbody>
<tr>
<td>fun f -&gt;</td>
<td>fun x -&gt;</td>
<td>not(f x);</td>
</tr>
<tr>
<td></td>
<td>fn</td>
<td></td>
</tr>
</tbody>
</table>

A function can also **take a function argument**

```plaintext
# let neg = fun f -> fun x -> not (f x);
... 
# let neg f x = not (f x);
val neg : int -> int -> bool = fn
# let is5gte = neg is5lt;
val is5gte : int -> bool = fn;
# is5gte 10;
val it : bool = false;
# is5gte 2;
val it : bool = true;
(*...odd, even ...*)
```

## A shorthand for function binding

```plaintext
# let neg = fun f -> fun x -> not (f x);
... 
# let neg f x = not (f x);
val neg : int -> int -> bool = fn
```

## Put it together: a “filter” function

If arg “matches” this pattern...  
...then use this Body Expr

```plaintext
- let rec filter f xs = 
  match xs with 
  | []      -> [] 
  | (x::xs')-> if f x 
    then x::(filter f xs') 
    else (filter f xs');;

val filter : ('a->bool) -> ('a -> bool) -> 'a list -> 'a list = fn
```

```plaintext
# let list1 = [1;31;12;4;7;2;10];
# filter is5lt list1;;
val it : int list = [31;12;7;10]
# filter is5gte list1;;
val it : int list = [1;4;2]
# filter even list1;;
val it : int list = [12;4;2;10]
```
Put it together: a “partition” function

```ocaml
# let partition f l = (filter f l, filter (neg f) l);
val partition :('a->bool)->'a list->'a list * 'a list =
fn
```

```ocaml
# let list1 = [1,31,12,4,7,2,10];
- ...
# partition is5lt list1 ;
val it : (int list * int list) = ([31,12,7,10],[1,2,10]
# partition even list1;
val it : (int list * int list) = ([12,4,2,10],[1,31,7])
```

A little trick ...

```ocaml
# 2 <= 3;; ...
val it : bool = true
# “ba” <= “ab”;;
val it : bool = false
```

```ocaml
# let lt = (<) ;;
val it : 'a -> 'a -> bool = fn
```

```ocaml
# lt 2 3;;
val it : bool = true;
# lt “ba” “ab” ;;
val it : bool = false;
```

```ocaml
# let is5Lt = lt 5;
val is5lt : int -> bool = fn;
# is5lt 10;
val it : bool = true;
# is5lt 2;
val it : bool = false;
```

Put it together: a “quicksort” function

```ocaml
let rec sort xs =
  match xs with
  | []     -> []
  | (h::t) -> let (l,r) = partition ((<) h) t in
              (sort l)@(h::(sort r))
```

Now, lets begin at the beginning ...