CSE 130 : Winter 2006
Programming Languages

Lecture 1: A Crash Course in ML

Ranjit Jhala
UC San Diego

Say hello to Standard ML

```ml
fun sort l[] = []
  | sort l (h::t) = let
    val (l,r) = List.partition (lt h) t
    in
      (sort l)@[h]@:(sort l r)
    end;
```

Quicksort in SML

Why readability matters...

```j
quicksort = :((#:<)&, #, #) ((#:<) ((#:<) :)) : (#:<) )
```

Quicksort in J

History, Variants

“Meta Language”

• Designed by Robin Milner @ Edinburgh
• Language to manipulate Theorems/Proofs
• Several dialects:
  - Standard ML (of New Jersey)
  - OCaml

News

• I’m substituting for Prof. Goguen
• 3 remaining programming assignments:
  - Out Thu, due next Fri
  - First one out tonight
• Office Hours: Tue after class
• Today: A crash course in ML

Say hello to Standard ML

```c
while (l != r - 1) {
    int l = beg + 1;
    int piv = arr[beg];
    int r = beg + 1;
    while (l != r - 1) {
        if(arr[l] == piv) { l++;
            else { swaps[1], save(--l);
        } if(arr[r] == piv || arr[r] > piv)
        { r--;
            else { if(arr[r] > piv || arr[r] < piv)
        { beg = l = r;
            else { swaps[1], save(beg);
            sort(arr, beg, mid);
            break;
        }
    }
}```

Quicksort in C
ML’s holy trinity

Expression → Value → Type

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

Interacting with ML

“Read-Eval-Print” Loop

Repeat:
1. System reads expression e
2. System evaluates e to get value v
3. System prints value v and type t

What are these expressions, values and types?

Base type: Integers

2;
2+2;
2 ♦ (9+10);  # 38
2 ♦ (9+10) -12;  # 26

Complex expressions using “operators”: (why the quotes?)
- +, -, *
- div, mod

Base type: Strings

“ab”;
“ab” ♦ “xy”;  ➔ “abxy”

Complex expressions using “operators”: (why the quotes?)
- Concatenation ♦

Base type: Booleans

true;
false;
1 < 2;
“aa” = “pq”;

(true) and also (1<2);  # false
(false) or else (1<2);  # true

Complex expressions using “operators”:
- “Relations”: =, <, <=, >,
- andalso, orelse, not

Type Errors

(2+3) orelse (“a” = “b”);

“pq” * 9;

(2 + “a”);

Untypable expression is rejected
- No casting or coercing
- Fancy algorithm to catch errors
- ML’s single most powerful feature
**Complex types: Product (tuples)**

- \((2+2 \ , 7>8)\) \rightarrow \((4,\text{false})\)

- \((9-3,\text{"ab"} \cdot \text{"cd"}, (2+2 \ , 7>8))\) \rightarrow \((6, \text{"abcd"}, \text{(4,\text{false})})\)

- Int * bool

**Complex types: Lists**

- 

- \([1, \text{"pq"}]\)

- All elements must have same type

**Complex types: Lists**

- List operator “Cons” ::

  - \(1::\text{[1]}\) \rightarrow \([1]\)
  - \(1::\text{[2]}\) \rightarrow \([1,2]\)
  - \(\text{"a"}::\text{["b","c"]}\) \rightarrow \([\text{"a"},\text{"b","c"]}\)

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

- \(1::[\text{"b"},\text{"cd"}]\)

**Complex types: Lists**

- List operator “Append” @

  - \([1,2]@[3,4,5]\) \rightarrow \([1,2,3,4,5]\)
  - \([\text{"a"}]@[\text{"b"},\text{"c"}]\) \rightarrow \([\text{"a"},\text{"b","c"}]\)

  - Can only append two lists

  - \(1@\text{[2,3]}\)

  - ... of the same type \(1@\text{[\"a\"\",\"b\""]}\)
Complex types: Lists

List operator “head”  \( \text{hd} \)

\[
\begin{align*}
\text{hd} \{1,2\} & \rightarrow 1 \quad \text{int} \\
\text{hd} \{a\} & \rightarrow a \quad \text{string}
\end{align*}
\]

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

Complex types: Lists

List operator “tail”  \( \text{tl} \)

\[
\begin{align*}
\text{tl} \{1,2,3\} & \rightarrow [2,3] \quad \text{int list} \\
\text{tl} \{a\} & \rightarrow [b] \quad \text{string list}
\end{align*}
\]

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

So far, a fancy calculator...

... what do we need next?

Variables and bindings

\[ \text{val} \ x = e; \]

“So bind the value of expression \( e \) to the variable \( x \)”

\[
\begin{align*}
- \text{val} \ x & = 2+2; \\
\text{val} \ x & = 4 : \text{int}
\end{align*}
\]

Variables and bindings

Later declared expressions can use \( x \)
- Most recent “bound” value used for evaluation

\[
\begin{align*}
- \text{val} \ x & = 2+2; \\
\text{val} \ x & = 4 : \text{int} \\
- \text{val} \ y & = x * x * x; \\
\text{val} \ y & = 64 : \text{int} \\
- \text{val} \ z & = [x,y,x+y]; \\
\text{val} \ z & = [4,64,68] : \text{int list}
\end{align*}
\]

Variables and bindings

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

\[
\begin{align*}
- \text{val} \ p & = a + 1; \\
\text{Error: unbound variable or constructor: a}
\end{align*}
\]

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

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

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

Binding by Pattern-Matching
Simultaneously bind several variables

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

Next: functions, but remember ...

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td></td>
</tr>
</tbody>
</table>

A function is a value!

Ok, it knows 1 now, but can it always ...?

Complex types: Functions!

```
fn x = x + 1
```

Parameter (formal)      Body  Expr
```
fn x = x + 1
```

```
fn
```

```
int -> int
```

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

A Problem

Can functions only have a single parameter?

<table>
<thead>
<tr>
<th>Parameter (formal)</th>
<th>Body  Expr</th>
</tr>
</thead>
<tbody>
<tr>
<td>fn x = x + 1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parameter (formal)</th>
<th>Body  Expr</th>
</tr>
</thead>
<tbody>
<tr>
<td>fn x = x + 1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>fn</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Parameter (formal)</th>
<th>Body  Expr</th>
</tr>
</thead>
<tbody>
<tr>
<td>fn x = x + 1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>fn</th>
</tr>
</thead>
</table>

How a call (“application”) is evaluated:
1. Evaluate argument
2. Bind formal to arg value
3. Evaluate “Body expr”
A Solution: Simultaneous Binding

Parameter | Body
---|---
\( f n \ x, y \) &gt; \( x &lt; y \) &gt; &gt; \( f n \) (int * int) &gt; bool

Can functions only have a single parameter?

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

Another Solution

Parameter | Body
---|---
\( f n \ x \) &gt; \( f n \ y \) &gt; \( x &lt; y \) &gt; &gt; \( f n \) (int &gt; int &gt; bool)

Whoa! A function can return a function

A shorthand for function binding

- \( \text{val neg} = f n \ f \) &gt; \( f n \ x \) &gt; \( \text{not} \ (f x) \) &gt; \( f n \) (int &gt; bool)
- \( \text{fun neg} \ f x \) &gt; \( \text{not} \ (f x) \) &gt; \( f n \) (int &gt; bool)

Put it together: a “filter” function

\(#\) If \( \text{arg} \) "matches"... then use \( \text{this pattern...} \) this Body Expr

- \( \text{fun filter f} \) \( \{ \} \) \( = \) \( \{ \} \)
  \( \mid \text{filter f} \) \( \langle \text{h::t} \rangle \) = \( \text{if} \) \( f \) \( \text{then} \) \( h \) \( \langle \text{filter f} \) \( t \rangle \rangle \)
  \( \text{else} \) \( \langle \text{filter f} \) \( t \rangle \rangle \)
- \( \text{val filter} = f n \) \( \langle \text{a→bool} \rangle \rightarrow \langle \text{a list} \rightarrow \text{a list} \rangle \)

- \( \text{fun list1} = [1,3,12,4,7,2,10] \); 
- \( \text{filter is5lt list1} \)
- \( \text{val list1} = [31,12,7,10] \) : int list

Put it together: a “partition” function

- \( \text{fun partition f} 1 \) \( = \) \( \langle \text{filter f} \) \( 1 \) \( \), \( \text{filter \ (neg f)} \) \( 1 \rangle \); 
- \( \text{val partition} = f n \langle \text{a→bool} \rangle \rightarrow \langle \text{a list} \rightarrow \text{a list} \rangle \)

- \( \text{fun list1} = [1,3,12,4,7,2,10] \); 
- \( \text{partition is5lt list1} \)
- \( \text{val list1} = [31,12,7,10] \); 
- \( \text{partition even list1} \)
- \( \text{val list1} = [1,3,7] \) : int list
Put it together: a “quicksort” function

```ocaml
fun sort lt [] = []
    | sort lt (h::t) =
        let
            val (l, r) = partition (lt h) t
        in
            (sort lt l)@(h:(sort lt r))
        end;
```

Next time: begin at the beginning ...