OCaml

The PL for the discerning hacker.

ML Flow

1. Enter expression
2. ML infers a type $\tau$
3. ML crunches expression down to a value
4. Value guaranteed to have type $\tau$

Expressions (Syntax) ➔ Exec-time “Dynamic” ➔ Values (Semantics)

Compile-time “Static” ➔ Types

ML Flow

Typing -> Eval Always Works

Complex types: Lists

- [];
- [1;2;3];
- [1;2;3];
- [1+1;2+2;3+3;4+4];
- ["a";"b"; "c";"d"];
- [(1, "a";"b") ; (3+4,"c")];
- [(1; "ab") ; (7,"c")];
- [(1;[2;3];[4;5;6])];
- [(1);[2;3];[4;5;6]]; 

'\ a list
int list
int list
int list
string list
(int*string) list
(int*string) list
(int list) list

- [1; "pq"];

All elements must have same type

- Unbounded size
- Can have lists of anything (e.g. lists of lists)
- But...
Question 1

Which of these causes a type error?

(a) [1; 2; 3]
(b) [“1”, 2, 3]
(c) “[1; 2; 3]”
(d) (1, 2, 3)
(e) [“1”; 2; 3]

Lists: Construct

Nil operator

[] : ’a list
[] => []

Cons operator

1::[2;3] : int list
[1;2;3]

e1:T e2:T list

Complex types: Lists

List operator “Cons” ::

1::[]; [1] int list
1::[2;3]; [1;2;3] int list
“a”::[“b”; “cd”]; [“a”; “b”; “c”] string list

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

1::[“b”; “cd”];

Complex types: Lists

List operator “Append” @

[1;2]@[3;4;5]; [1;2;3;4;5] int list
[“a”]@[“b”]; [“a”; “b”] string list
[]@[1]; [1] int list

Can only append two lists of the same type

1 @ [2;3];

[1]@[“a”; “b”];
Complex types: Lists

List operator “head”  hd

<table>
<thead>
<tr>
<th>Operator</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>hd [1;2];</td>
<td>int</td>
</tr>
<tr>
<td>hd ([&quot;a&quot;],[&quot;b&quot;]);</td>
<td>string</td>
</tr>
</tbody>
</table>

Only take the head a nonempty list  hd [];

Complex types: Lists

List operator “tail”  tl

<table>
<thead>
<tr>
<th>Operator</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>tl [1;2;3];</td>
<td>int list</td>
</tr>
<tr>
<td>tl ([&quot;a&quot;],[&quot;b&quot;]);</td>
<td>string list</td>
</tr>
</tbody>
</table>

Only take the tail of nonempty list  tl [];

Question 2: What is result of?

(hd [[];[1;2;3]]) = (hd [][];[“a”]);

(a) Syntax Error
(b) true : bool
(c) false : bool
(d) Type Error (hd)
(e) Type Error (=)

Lists: Deconstruct

Head

<table>
<thead>
<tr>
<th>e : T list</th>
<th>e =&gt; v1::v2</th>
</tr>
</thead>
<tbody>
<tr>
<td>hd e : T</td>
<td>hd e =&gt; v1</td>
</tr>
</tbody>
</table>

Tail

<table>
<thead>
<tr>
<th>e : T list</th>
<th>e =&gt; v1::v2</th>
</tr>
</thead>
<tbody>
<tr>
<td>tl e : T list</td>
<td>tl e =&gt; v2</td>
</tr>
</tbody>
</table>

(hd [[];[1;2;3]]) = (hd [][];[“a”]);

<table>
<thead>
<tr>
<th>e1 : T list</th>
<th>e1 =&gt; v1::v2</th>
</tr>
</thead>
<tbody>
<tr>
<td>e2 : bool</td>
<td>e2 = e1 : bool</td>
</tr>
</tbody>
</table>
Recap: Tuples vs. Lists?

What’s the difference?

- **Tuples:**
  - Different types, but **fixed** number:
    - pair = 2 elts
    - triple = 3 elts
  - Syntax:
    - Tuples = *comma*
    - Lists = *semicolon*

- **Lists:**
  - **Same type**, **unbounded** number:
    - Syntax:
      - Tuples = *comma*
    - Lists = *semicolon*

So far, a fancy calculator...

... what do we need next?

**Question 3: What is result of?**

\[
\text{if (1 < 2) then true else false}
\]

(a) Syntax Error
(b) true
(c) false
(d) Type Error
Question 4: What is result of?

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

(a) Syntax Error
(b) [1;2]
(c) 5
(d) Type Error

If-then-else expressions

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

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

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

(int list) = (string list)

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

If (1 < 2) then [1;2] else 5

if false then [1;2] else 5

\[
e_1 : \text{bool} 
\]

\[
e_2 : T 
\]

\[
e_3 : T 
\]

\[
\text{if } e_1 \text{ then } e_2 \text{ else } e_3 : T
\]

So far, a fancy calculator...

Variables
Question 5: I got this @ prompt

```ocaml
# [x+x; x*x] ;;
- : int list = [20; 100]
```

What had I typed before?
(a) \( x = 10; \)
(b) `int x = 10;`
(c) \( x == 10; \)
(d) `let x = 10;`
(e) \( x := 10; \)

---

**Variables and bindings**

```ocaml
let \( x = e ;;\)
```

"Bind the value of expression \( e \) to the variable \( x \)"

```ocaml
# let x = 2+2;;
val x : int = 4
```

---

Variables and bindings

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

```ocaml
# 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]
```

---

Variables and bindings

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

```ocaml
# 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

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

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

Question 6: What is result of?

```ocaml
let x = 10 in
(let z = 10 in x + z) + z
```

(a) Syntax Error
(b) 30
(c) Unbound Error -- x
(d) Unbound Error -- z
(e) Type Error

Binding by Pattern-Matching

Simultaneously bind several variables

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

Why is it whining?

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

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

Another useful early warning
Binding by Pattern-Matching

But what of:

```ocaml
# 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?

```ocaml
# let h::t = [];
Exception: Match_failure
```

```ocaml
# 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: NEVER USE PATTERN MATCHING LIKE THIS

ALWAYS USE THIS FORM INSTEAD

```ocaml
match l with ...
```

(coming up soon, but this is important)

Functions up now, remember ...

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

A function is a value!

Complex types: Functions!

```ocaml
fun x -> x+1;;
fn
```

```ocaml
# let inc = fun x -> x+1 ; val inc : int -> int = fn
# inc 0;
val it : int = 1
# inc 10;
val it : int = 11
```
A Problem

Functions only have ONE parameter ?!

Parameter (formal) | Body Expr
---|---
fun 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 Solution: Simultaneous Binding

Parameter (formal) | Body Expr
---|---
fun (x,y) -> x<y; | fn
(int * int) -> bool

Functions only have ONE parameter ?!

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

Another Solution ("Currying")

Parameter (formal) | Body Expr
---|---
fun x -> fun y-> x<y; | fn
int -> (int -> bool)

Whoa! A function can return a function

Question 7: What is result of?

(fun x -> not x)

(a) Syntax Error
(b) <fun> : int -> int
(c) <fun> : int -> bool
(d) <fun> : bool -> int
(e) <fun> : bool -> bool
And how about…

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

A function can also take a function argument

```
# let neg = fun f -> fun 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

```
# 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

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

# let list1 = [1;31;12;4;7;2;10];
# filter is5lte 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]

Question 8: What is result of?

```
(fun f -> (fun x -> (f x)+x))
```

(a) Syntax Error
(b) Type Error
(c) <fun> : int -> int -> int
(d) <fun> : int -> int
(e) <fun> : (int->int)-> int -> int
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];
...
```

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

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

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 ...

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

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