OCaml

The PL for the discerning hacker.

ML Flow

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

Complex types: Lists

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

<table>
<thead>
<tr>
<th>[];</th>
<th>[];</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1;2;3];</td>
<td>[1;2;3];</td>
</tr>
<tr>
<td>[1+1;2+2;3+3;4+4];</td>
<td>[2;4;6;8];</td>
</tr>
<tr>
<td>[&quot;a&quot;;&quot;b&quot;; &quot;c&quot;;&quot;d&quot;]</td>
<td>[&quot;a&quot;;&quot;b&quot;; &quot;cd&quot;]</td>
</tr>
<tr>
<td>[(1,&quot;a&quot;,&quot;b&quot;); (3+4,&quot;c&quot;)];</td>
<td>[(1,&quot;ab&quot;);(7,&quot;c&quot;)];</td>
</tr>
<tr>
<td>[[1];[2;3];[4;5;6]];</td>
<td>[[1];[2;3];[4;5;6]];</td>
</tr>
</tbody>
</table>

All elements must have same type

Complex types: Lists

- 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]

Complex types: Lists

List operator “Cons” ::

1::[];

[1]

int list

1::[2;3];

[1;2;3]

int list

“a”::[“b”;“c”];

[“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];

[1]

int list

Can only append two lists

1 @ [2;3];

... of the same type

[1] @ [“a”;“b”];
Complex types: Lists

List operator “head”  \( \text{hd} \)

\[
\begin{align*}
\text{hd} \ [1;2]; & \quad 1 \quad \text{int} \\
\text{hd} \ (\text{"a"} @ \text{"b"}); & \quad \text{"a"} \quad \text{string}
\end{align*}
\]

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

Complex types: Lists

List operator “tail”  \( \text{tl} \)

\[
\begin{align*}
\text{tl} \ [1;2;3]; & \quad [2;3] \quad \text{int list} \\
\text{tl} \ (\text{"a"} @ \text{"b"}); & \quad [\text{"b"}] \quad \text{string list}
\end{align*}
\]

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

Question 2: What is result of?

\((\text{hd} \ [ ]; [1;2;3])) = (\text{hd} \ [ ]; \text{"a"})\)

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

Lists: Deconstruct

Head

\[
\begin{align*}
\text{e : T list} & \quad \text{e => v1::v2} \\
\text{hd} \ e : T & \quad \text{hd} \ e => v1
\end{align*}
\]

Tail

\[
\begin{align*}
\text{e : T list} & \quad \text{e => v1::v2} \\
\text{tl} \ e : T list & \quad \text{tl} \ e => v2
\end{align*}
\]

\((\text{hd} \ [ ]; [1;2;3])) = (\text{hd} \ [ ]; \text{"a"})\)

\[
\begin{align*}
\text{e : T list} & \quad \text{e => v1::v2} \\
\text{e1} = \text{e2 : bool} & \quad \text{string list}
\end{align*}
\]
Recap: Tuples vs. Lists?

What’s the difference?

- **Tuples:**
  - Different types, but **fixed** number:
    - pair = 2 elts
      - (3, “abcd”) (int * string)
    - triple = 3 elts
      - (3, “abcd”, (3.5, 4.2)) (int * string * (float * float))

- **Lists:**
  - **Same** type, **unbounded** number:
    - [3; 4; 5; 6; 7] int list

- **Syntax:**
  - Tuples = comma      Lists = semicolon

So far, a fancy calculator...

... what do we need next?

So far, a fancy calculator...

Branches

Question 3: What is result of?

if (1 < 2) then true else false

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

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

So far, a fancy calculator...

Variables

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

```
let x = e;;
```

“Bind the value of expression `e` to the variable `x`”

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

---

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]
#
```

---

Variables and bindings

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

Question 6: What is result of?

```
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

```
# 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
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
```

Another useful early warning

Functions up now, remember ...

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

A function is a value!

Complex types: Functions!

```
fun x -> x+1);
```

```
val inc : int -> int = fn
```

```
# inc 0;
val it : int = 1
```

```
# inc 10;
val it : int = 11
```

(coming up soon, but this is important)
A Problem

Functions only have ONE parameter ?!

Another Solution (“Currying”)

Whoa! A function can return a function

Question 7: What is result of?

(a) Syntax Error
(b) <fun> : int -> int
(c) <fun> : int -> bool
(d) <fun> : bool -> int
(e) <fun> : bool -> bool
A shorthand for function binding

A function can also take a function argument

A shorthand for function binding

Question 8: What is result of?

\[(\text{fun } f \to (\text{fun } x \to (f x) + x))\]

(a) Syntax Error
(b) Type Error
(c) \(<\text{fun}> : \text{int} \to \text{int} \to \text{int}\>
(d) \(<\text{fun}> : \text{int} \to \text{int}\>
(e) \(<\text{fun}> : (\text{int}\to\text{int}) \to \text{int} \to \text{int}\>

Put it together: a “filter” function

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

- 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 list->'a list) = fn

# 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

# 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
# let lt = (<) ;;
val it : 'a -> 'a -> bool = fn
# lt 2 3;;
val it : bool = true;
# lt "ba" "ab" ;;
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 ...

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

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