OCaml

The PL for the discerning hacker.
Hello! My name is Zach.

I’ll be your guide for today.
MOVE FAST AND BREAK THINGS
1. Enter expression
2. ML infers a type $\tau$
3. ML crunches expression down to a value
4. Value guaranteed to have type $\tau$
Complex types: Lists

- Unbounded size
- Can have lists of anything (e.g. lists of lists)
- But...
Complex types: Lists

All elements must have same type

[1; "pq"];
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]
Complex types: Lists

List operator “Cons” ::

1::[]; [1]

1::[2;3]; [1;2;3]

“a”::[“b”;“c”]; [“a”;“b”;“c”]

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

1::[“b”;“cd”];
Lists: Construct

Nil operator

\[ [] \]
\[ [] : \text{'a list} \]
\[ [] => [] \]

Cons operator

\[ 1 :: [2; 3] \]
\[ \text{int list} \]
\[ 1; 2; 3 \]
\[ \underline{e1 : T} \underline{e2 : T list} \]
\[ e1 :: e2 : T list \]
\[ \underline{e1 = v1} \underline{e2 = v2} \]
\[ e1 :: e2 => v1 :: v2 \]
Complex types: Lists

List operator “Append”  @

\[[1;2]@[3;4;5];\]  \[[1;2;3;4;5]\]  int list

\[\text{"a"}@[\text{"b"}];\]  \[\text{"a"};\text{"b"}\]  string list

\[[\text{]}@[\text{]};\]  \[[\text{]}\]  int list

Can only append two lists ... of the same type

\[1 @ [2;3];\]

\[\text{[1]} @ [\text{"a"};\text{"b"}];\]
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” \textbf{tl}

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

Only take the tail of nonempty list \textbf{tl} [ ];
Question 2: What is result of?

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

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

Head

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

Tail

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

(int list)

\[
(hd \ [[[];[1;2;3]]]) = (hd \ [[[];[“a”]]])
\]

(string list)

\[
e_1:T \quad e_2:T \\
e_1=e_2 : \text{bool}
\]
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?

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

(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

\[
\begin{align*}
e_1 &: \text{bool} \\
e_2 &: T \\
e_3 &: T
\end{align*}
\]
\[
\text{if } e_1 \text{ then } e_2 \text{ else } e_3 : T
\]
If-then-else expressions

- `if (1 < 2) then [1;2] else 5`
- `if false then [1;2] else 5`

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

- `e1 : bool  e2 : T  e3 : T`
- `if e1 then e2 else e3 : T`
So far, a fancy calculator...

Variables
Question 5: I got this @ prompt

```coconut
# [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

```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!

```
# 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

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

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

\[17424 \quad \text{int}\]
Question 6: What is result of?

```plaintext
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]
```
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
# 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:

```ml
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?

```ml
let h::t = [];
```

Exception: Match_failure

```ml
let XS = [1;2;3];
val xs = [1;2;3]: list
```

```ml
let 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

```
let h::t = ...
```

ALWAYS USE THIS FORM INSTEAD

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

(coming up soon, but this is important)
Functions
Functions up now, 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
--- | ---
\texttt{fun x -> x+1;} | \texttt{fn}

\texttt{fun x -> x+1;} \rightarrow \texttt{fn}

\texttt{let inc = fun x -> x+1 ;}
\texttt{val inc : int -> int = fn}
\texttt{# inc 0;}
\texttt{val it : int = 1}
\texttt{# inc 10;}
\texttt{val it : int = 11}
A Problem

Functions only have ONE parameter?!

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

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

Functions only have ONE parameter?
Another Solution ("Currying")

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

int -> (int -> bool)

Whoa! A function can return a function

```ocaml
# 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 ;
```
Question 7: What is result of?

(func \ x \ -> \ not \ x)\n
(a) Syntax Error
(b) func : int -> int
(c) func : int -> bool
(d) func : bool -> int
(e) func : bool -> bool
And how about…

Parameter | Body
---|---
(formal) | Expr

fun \( f \rightarrow \text{fun } x \rightarrow \text{not}(f\ x) \);

\((\text{'a } \rightarrow \text{bool}) \rightarrow (\text{'a } \rightarrow \text{bool})\)

A function can also take a function argument

```ocaml
# let neg = fun f -> fun x -> not (f x);
val lt : 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 ...*)
```
Question 8: What is result of?

\((\text{fun } f \rightarrow (\text{fun } x \rightarrow (f \ x) + x))\)

(a) Syntax Error
(b) Type Error
(c) \(<\text{fun}> : \text{int} \rightarrow \text{int} \rightarrow \text{int}\>
(d) \(<\text{fun}> : \text{int} \rightarrow \text{int}\>
(e) \(<\text{fun}> : (\text{int} \rightarrow \text{int}) \rightarrow \text{int} \rightarrow \text{int}\>
A shorthand for function binding

```ocaml
# 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;
```
Put it together: a “filter” function

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

```ml
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 lisi) = 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 ...

# 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;

# 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

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

Now, let’s begin at the beginning ...