OCaml

The PL for the discerning hacker.

ML Flow

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

Expressions

Values

Types

Compile-time

“Static”

Exec-time

“Dynamic”

Typing -> Eval Always Works

Complex types: Lists

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

All elements must have same type

Complex types: Lists
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” \(::\)

List operator “Append” \(@\)

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

\(1::[“b”; “cd”];\)

Can only append two lists \(...\) of the same type

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

Lists: Construct

Nil operator

\[
[]:
\[\]

Cons operator

\[
1::[2;3]:
\[1;2;3]\n\]

\[
e1:T\ e2:T\ list\ e1::e2:T\ list\ e1::e2:v1::v2
\]
Complex types: Lists

List operator “head” \( \text{hd} \)

- \( \text{hd} \ [1;2]; \) \( \text{int} \)
- \( \text{hd} \ \{[“a”];[“b”]\}; \) \( \text{“a”} \)

Only take the head a nonempty list

\( \text{hd} []; \)

Complex types: Lists

List operator “tail” \( \text{tl} \)

- \( \text{tl} \ [1;2;3]; \) \( \text{int list} \)
- \( \text{tl} \ \{[“a”];[“b”]\}; \) \( \text{[“b”]} \)

Only take the tail of nonempty list

\( \text{tl} []; \)

Question 2: What is result of?

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

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

Lists: Deconstruct

Head

\( e \ : \text{T list} \)
\( \text{hd} \ e \ : \text{T} \)
\( e \ => \text{v1}::\text{v2} \)

Tail

\( e \ : \text{T list} \)
\( \text{tl} \ e \ : \text{T list} \)
\( e \ => \text{v1}::\text{v2} \)

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

\( \text{e}_1 \::\text{T list} \)
\( \text{e}_2 \::\text{T list} \)
\( \text{e}_1 = \text{e}_2 : \text{bool} \)
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?

Question 3: What is result of?

if (1 < 2) then true else false

(a) Syntax Error
(b) true
(c) false
(d) Type Error

Branches
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

(if 1>2 then [1,2] else []) = (if 1<2 then [] else ["a"])

So far, a fancy calculator...

Variables
Question 5: I got this @ prompt

```haskell
# [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
- **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]
```

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:

```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

ALWAYS USE THIS FORM INSTEAD

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

(coming up soon, but this is important)

Functions

Functions up now, remember ...

Expression  Value

Expression is an expression
Expression has a value
Expression has a type

Type

A function is a value!

Complex types: Functions!

Parameter

(formal)

Body

Expr

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

fn

```
int -> int
```

```
# 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 ?!

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
And how about…

A function can also take a function argument

```haskell
# 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

```haskell
# let neg f x = not (f x);
val neg : int -> int -> bool = fn
# let is5gte f x = not (f x);
val is5gte : int -> bool = fn
```

Put it together: a “filter” function

```haskell
- let rec filter f xs =     match xs with 
    | []      -> []    | ... 
val filter : (a->bool)->'a list->'a list) = fn
```

Question 8: What is result of?

```haskell
(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

# 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;

# 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 ...