Lecture 2

Monday, August 23, 2021 5:53 PM

- Ocaml vs. Haskell vs. Scala

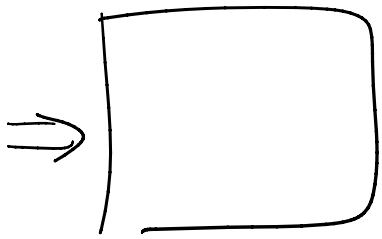
Ocaml ↓ Eager evaluation

Haskell ↓ Lazy evaluation

Scala ↓ JVM ↓ Easy to fall into the trap of using Scala to transcribe Java code

---

- First-class fns
- Static types
- ...



Statics

How is the program compiled.

- Basic types
- Type checking
- Type inference
  - Generics
    - Parametric polymorphism

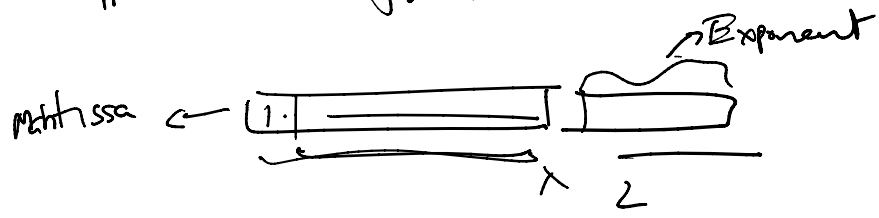
Dynamics

How does the program execute?

- Name resolution
- Let bindings
- Eval order

Basic types: int float char

Basic types: int float char



string

bool

Short circuit evaluation $\left\{ \begin{array}{l} e_1 \;\&\&\; e_2 \iff \text{if } (e_1) \text{ then } e_2 \text{ else false} \\ e_1 \;||\; e_2 \iff \text{if } (e_1) \text{ then true else } e_2 \end{array} \right.$

---

unit $\longleftrightarrow$ Nothing (Scala)

type with 1 value

type with 0 values

Lists

tuples
- fixed size

Functions let f x = x + 1

fun x -> x+1

SKI combinator (\x -> x+1, Haskell)

SKI combinator calculus

(\x → x+1, Haskell)

(λx . x+1, Lambda calculus)

Type checking

- foo e_1 e_2

① Find type of foo

② Find type of e_1 T_1

③ Find type of e_2 T_2

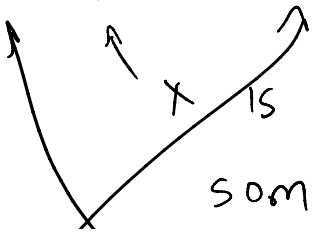
④ Confirm that foo : $T_1 \to T_2 \to T_3$

⑤ The whole expression then has type T_3

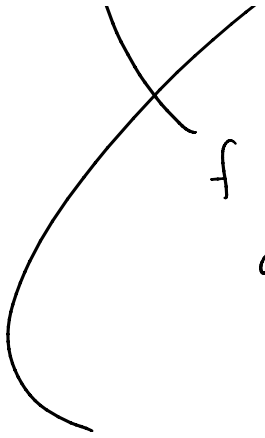
Let expressions

Type inference

let f x = x ;



x is a variable. It must have some type 'a



some type 'a

f is a function. It takes x as input. So it must have type 'a → 'b, for some 'b.

f x returns x as output.

So 'b = 'a

---

x : 'a f : 'a → 'a

How to evaluate "things"

Ground values: 3, true, [1; 2; 8] etc.

Do nothing

Let expressions:

let x = e_1 in e_2

① Evaluate e_1. Say it produces v_1.

① Evaluate e_1. Say it produces v_1.

② In e_2, replace all unshadowed occurrences of x with v_1 (Say e_2')

③ Evaluate e_2'.

If expressions

if e_1 then e_2 else e_3

① Evaluate e_1. Say result is v_1.
Type checker guarantees that v_1 is a bool.
So v_1 = true or v_1 = false.

② If v_1 = true, then evaluate e_2. Say v_2.
Return v_2

③ Otherwise evaluate e_3. Say v_3.
Return v_3.

Function definitions:

fun $x \rightarrow x + 5$

Do nothing

Do nothing

Function calls (Eager)

e_1 e_2

① Finish evaluating e_1 into v_1

& e_2 into v_2

② The type checker has guaranteed that v_1 is a function which can take v_2 as input.

③ Say that v_1 = fun $x \rightarrow$ e_3

In e_3, replace all (unshadowed) occurrences of x with v_2. Say e_3'

④ Evaluate e_3'

Alternative approach
Lazy: Evaluate e_1 first. Evaluate e_2 when needed, if needed (Haskell)

$(\text{fun } x \to x+3)\ (3+4)$

$\Rightarrow (\text{fun } x \to x+3)\ 7$

$\Rightarrow 7+3$

$\Rightarrow 10$

Q: Can you define if-then-else as a function?

if e_1 then e_2 else e_3

ite e_1 e_2 e_3

In Ocaml

let ite e_1 e_2 e_3 = if e_1 then e_2 else e_3

ite true (3+4)  ⇒ ite true 7 

In Haskell

ite true (3+4) crash

⇒ if true then (3+4) else crash

⇒ 3+4

⇒ 7

Let expressions vs. function calls

let $x = e_1$ in e_2 ⟺

(fun x → e_2)(e_1)