Issue: we are performing code transformation each step

↳ Compiled Programming Languages (C++, C, Rust)

↳ the thing being executed: byte code (Java)

↳ Memory (Static, Program, Stack, Heap)

↳ Bad!

Read only

↳ JIT (Just-in-time-compiled) (JS, Julia)

↳ bytecode. ← optimization in V8 ← Google

↳ Good!

↳ Hard to get a JIT engine

JS engine

Webkit ← Apple

... ← Mozilla

↳ Interpreted (Python)

↳ AST manipulation

↳ okay but slow

```
Let add = Fun x -> Fun y -> x + y In
Let f1 = add 1 In
Let f2 = add 3 In
...
```

← add multiple times

2. What is Closure:

$(\lambda x.\ e\ ,\ \rho)$

$\lambda x.\ e$: Function

ρ: Environment (who) — eval_env. $\{x \mapsto v\}$

add = Fun x → [e: Fun y → x+y]

FV(e) = {x}

e is not closed

add 3 → (Fun y → x+y , { x ↦ 3 })

add 4 → (Fun y → x+y , { x ↦ 4 })

Same

different

Code is not changed

Closure: $(\lambda x.e\ ,\ \rho)$ where

$\rho = \{ y_1 \mapsto v_1 , y_2 \mapsto v_2 , \dots , y_n \mapsto v_n \}$

$\{y_1, \dots, y_n\} = FV(\lambda x.e)$

$(\lambda x.e, \rho)\ v \Rightarrow e[v_i/y_i \text{ for } y_i \mapsto v_i \text{ in } \rho][v/x]$

JavaScript: Interpreted ← Handle during runtime

C++: Compiled ← allow user to specify more

Rust: Compiled ← Type Inference Engine

(CPS) Continuation Passing Style Programming

Javascript:

```
let add = (x) => {
    return (y) => {
        return x+y;
    }
}
add(3);
```

$((y) \Rightarrow \{ \text{return } x+y \}, \{x \mapsto 3\})$

1. dynamically we collect the free variables in the scope

closure environment is just storing those that are used.

2. function pointer themselves are also in the env just under a different group (scripts)

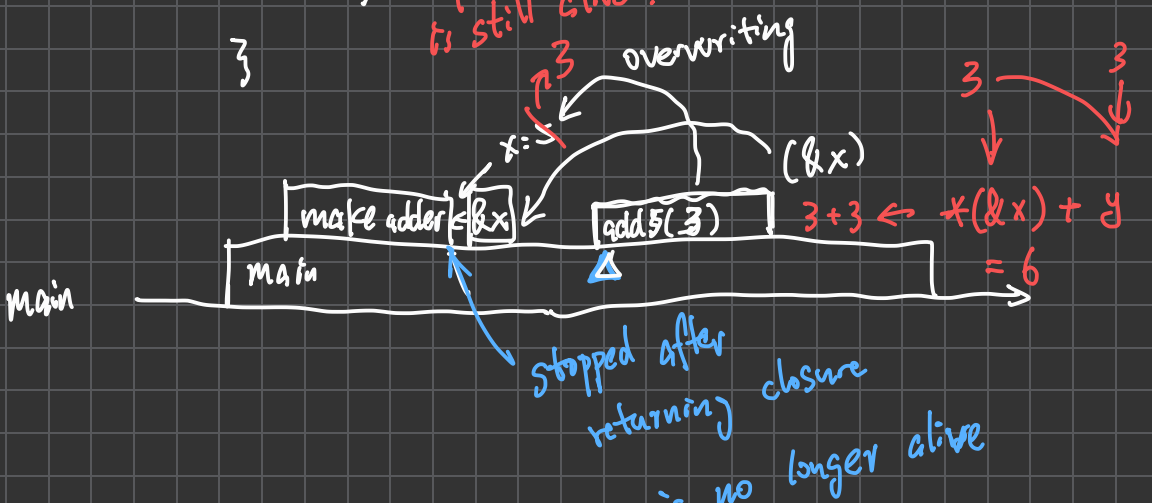
3. variable liveliness ← Garbage collection (GC) Handles

C++

```
make_adder2 (int x) {
    [&x] (int y) {
        return x + y;
    }
}
```

lofc time

is still alive?



stopped after returning closure

x is no longer alive

C++ Summary:

1. Without compiler help, you as programmer need to specify captured variables (free variables)

   and for each variable, specify either

   a. capture by ref ← save the ref into env

   b. capture by value ← copy the value into env

2. When we use capture by ref, we need to pay attention to liveliness of the variable. We want the captured reference to live longer than all the places where the closure is invoked

[&] ← auto ...

Rust :

1. Automatically capture by reference
2. Has liveliness analysis so that program is memory safe

   make sure that

   captured ref outlive closure