CSE 231: Closures

Ranjit Jhala *May 12, 2026*

Closures

In a few steps, we will extend our language with the ability to treat functions as first-class values.

1. Labels
2. Anonymous functions
3. Arity
4. Free Variables
5. Recursion

Part 1: Functions as Values

QUIZ: Semantics

What should the following code evaluate to?

```
(fun (f it)
  (it 5))

(fun (inc x)
  (+ x 1))

(f inc)
```

Abstract Syntax

```
pub struct Defn {
  name: String,
  params: Vec<String>,
  body: Expr,
}

pub enum Expr { ...
    Fun(Defn),
}
```

Parser

No need for `Prog`! But need to change the parser, to produce a single `Expr` instead of a `Prog`.

```
fn prog(defns: Vec<Defn>, e: Expr) -> Expr {
  ...
}
```

Evaluator

The type of `Val` is extended to include `Fun`

```
pub enum Val { ...
    Fun(Defn),
}
```

QUIZ: Extend `eval` to remove `funs`

```
fn eval(&self, e: &Expr, env: &mut Env) -> Val {
  match e {
    ...
    Expr::Defn(defn) =>
      _____________________________,
    Expr::Call1(f, e1) => {
      let v1 = eval(e1, env)?;

      _____________________________________________

      _____________________________________________

      _____________________________________________

      _____________________________________________

      _____________________________________________

    }
    ...
  }
}
```

QUIZ: Compiler: Fun Call

```
Fun(defn) => self.compile_defn(defn),

Call1(f, e1) =>

  _____________________________________________

  _____________________________________________

  _____________________________________________

  _____________________________________________
```

Compiler: Fun Definition

```
fn compile_defn(&mut self, defn: &expr::Defn) ->
String {
  let name = defn.name.clone();
  // labels
  let entry_label = entry_label(&name);
  let finish_label = finish_label(&name);
  let body_label = body_label(&name);
  // setup, body, exit
  let entry = compile_entry(&defn.body);
  let stack = Stack::new(&defn.params[..]);
  let body = self.compile_expr(&defn.body, &stack,
  &exit_label());
  let exit = compile_exit();
  // code
  format!(
    "jmp {finish_label}\n\
     {entry_label}:\n\
     {entry}\n\
     {body_label}:\n\
     {body}\n\
     {exit}\n\
     {finish_label}:\n\
     lea rax, [rel {entry_label}]"
  )
}
```

QUIZ: Stack Allocation

What is `max_vars` for a `Defn`?

Part 2: Anonymous functions

Since `fun` are also `let`-bound values, we can just use `let` to name them. So we can write:

QUIZ: Semantics

What should the following code evaluate to?

```
; anon function that takes `it` and returns `it 5`
(let (f   (fn (it) (it 5)))

; anon function that takes `z` and returns `(+ z 1)`
(let (inc (fn (z) (+ z 1)))

  (f inc)

))
```

Abstract Syntax

We just need to support **nameless** definitions, so we can reuse `Defn` with an optional name.

```
pub struct Defn {
  name: Option<String>,
  params: Vec<String>,
  body: Expr,
}
```

Update the parser to support `fn` as well as `fun`...

Evaluation, Compilation is unchanged!

Part 3: Arity

But wait a minute... what if we call a function with the wrong number of arguments?

QUIZ: Semantics What should the following code evaluate to?

```
(let (f   (fn (it)    (it 5)))

(let (add (fn (x1 x2) (+ x1 x2)))

  (f add)

))
```

Representation

Have to track the *arity* of each function, i.e. the number of parameters it takes.

Store a **function value** as **pointer** to a `Vec`-like structure

- arity (at [offset 0])
- code-label (at [offset 8])

```
[fun-arity][fun-label]
```

Problem need to *distinguish* this from *other* values, e.g. `int` or `bool` or `vec`!

Tags

Modify tagging scheme for `vec` to use the `0101` tag for function values.

Leaves **other** types unchanged.

Value	Representation
`number`	`xxx0`
`false`	`x011`

```
true          x111
nil           1001
(vec ...)     0001
(fun ...)     0101
```

Runtime Types

```
pub enum Ty { ...
  Fun,
}

const CLOSURE: i64 = 5;

fn test_type(ty: Ty) -> String {
  let (mask, expected) = match ty {
    Ty::Num => (0b1, 0),
    Ty::Bool => (0b11, 3),
    Ty::Vec => (0b111, 1),
    Ty::Fun => (0b111, 5),
  };
  format!(
    "mov rcx, rax\n\
     and rcx, {mask}\n\
     cmp rcx, {expected}\n\
     mov rcx, {code}\n\
     cmovne rdi, rcx\n\
     jne label_error"
  )
}
```

Compiler: Call

```
fn compile_callee(f: &String, arity: usize) ->
String {
  ...
  format!(
    "; load closure point\n\
     mov rax, [rbp - 8*{f_pos}]\n\
     {check_closure}\n\
     ; strip tag\n\
     mov rcx, rax\n\
     sub rcx, {CLOSURE}\n\
     ; check arity\n\
     mov rcx, [rcx]\n\
     cmp rcx, {arity}\n\
     mov rdx, 77\n\
     cmovne rdi, rdx\n\
     jne label_error")
```

```
  )
}
```

```
fn compile_expr(e: &Expr, stack: &Stack) -> String {
  match e {
    Call1(f, e1) => {
      let e1_code = self.compile_expr(e1, stack,...);
      let f_code = self.compile_callee(f, stack, 1);
      format!(
        "{e_code}\n\
         push rax\n\
         {f_code}\n\
         sub rax, {CLOSURE}\n\
         call rax\n\
         add rsp, 8")
    }
  }
}
```

Compiler: Definition

```
fn compile_defn(&mut self, defn: &expr::Defn) ->
String {
    "jmp {finish_label}\n\
     {entry}\n\
     {body_label}:\n\
     {body}\n\
     {exit}\n\
     {finish_label}:\n\
     ; write arity \n\
     mov rcx, {arity}\n\
     mov [r11], rcx\n\
     ; write label \n\
     lea rcx, [rel {entry_label}]\n\
     mov [r11 + 8], rcx\n\
     ; set closure address\n\
     mov rax, r11\n\
     add r11, 16\n\
     add rax, {CLOSURE}"
}
```

Part 4: Free Variables

Part 5: Recursion