

CSE 231: Functions as Values in Asm

Ranjit Jhala May 14, 2026

Closures

Lets compile functions as **first-class values**. The key challenge: what is the *the value of a function*?

Problem 1: Definitions

What is *value* of rax after executing `< (fun (f x) e) > ?`

Problem 2: Calls

How can we *use* the value to *call* a function value ?

Example 00: Let-binding Functions

Lets write assembly for

```
(let (inc                                ; 2. BIND-TO-NAME
      (fun (inc_def x) (add1 x))        ; 1. DEFINITION
      )
      (inc 5))                          ; 3. CALL
```

At a high level, the assembly will look like this:

```
our_code_starts_here:
push rbp                                ; -----STACK SETUP
mov rbp, rsp
sub rsp, 8*1000
mov [rbp - 8*1], rdi ; save input
mov r11, rsi ; save heap-pointer

; 1. DEFINITION <(fun (inc x) (add1 x))>

; 2. BIND-TO-NAME <(let (inc ...) ...)>

; 3. CALL <(inc 5)>

our_code_ends_here: ; ----- STACK TEARDOWN
mov rsp, rbp
pop rbp
ret
```

Idea 1: Function = Code Label

Idea Lets try to use the *code label* of the function as its value.

1. **Define** compile code for `inc`, save *entry-label* in `rax`;
2. **Bind** by saving value of `rax` at stack slot for `inc`;
3. **Call** By call-ing the label stored at stack slot for `inc`.

QUIZ: Definition

Assembly for `<(fun (inc_def x) (add1 x))>`

```

; DEFINITION for `(fun (inc_def x) (add1 x))>`

fun_start_inc_def:
push rbp
mov rbp, rsp
sub rsp, 8*1000

mov rsp, rbp
pop rbp
ret

```

Example: Bind-to-Name

Assembly for `<let (inc ...)>`

```

mov [rbp - 8*2], rax ; `inc` @ stack slot 2

```

QUIZ: Call

Assembly for `<(inc 5)>`

Example 01: Nameless Functions

Why **name** the `inc_def` function? How would `asm` change?

```

(let (inc ; 2. BIND-TO-NAME
      (fn (x) (add1 x)) ; 1. ANONYMOUS-DEFINITION
      )
  (inc 5)) ; 3. CALL

```

Example 02: Code Label is Not Enough!

Lets apply our strategy to the below code.

```
(let (inc                                ; 2. BIND-TO-NAME;
      (fn (x y) (+ x y))                ; 1. ANONYMOUS-DEFINITION
      (inc 5))                           ; CALL
```

Or even this code

```
(let (inc 0)
      (inc 5))
```

What might go wrong?

What *other* information is needed about `inc`'s **definition**?

Idea 2: Function = (Arity, Code Label)

When compiling a call `(f e1 ... en)` we need to know

1. That `f` is a *function*, and not, e.g. a number,
2. That the function's *arity* equals the number of arguments.

Idea Represent function as a pair of its *arity* and *code label*.

arity	code-label
-------	------------

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

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

Problem how to *distinguish* from *other* vec-like values?

Tags

Modify last-4-bits tagging scheme to use 0101 for functions.

(**Other** types unchanged.)

Value	4 LSB
number	xxx0
false	x011
true	x111
nil	1001
(vec ...)	0001
(fun ...)	0101

Runtime Type Checking

Lets recompile using our new representation

```
(let (inc ; 2. BIND-TO-NAME;
      (fn (x y) (+ x y)) ; 1. ANONYMOUS-DEFINITION
      (inc 5)) ; CALL
```

The high-level assembly will still be the same

```
; 1. DEFINITION <(fn (x y) (+ x y))>
; 2. BIND-TO-NAME <(let (inc ...) ...)>
; 3. CALL <(inc 5)>
```

QUIZ: Definition using (Arity, Code Label)

Complete the asm for <(fn (x y) (+ x y))>.

Use r11 to fill in the arity and code-label fields.

Then set the tag before assigning into rax.

```
; 1. CODE FOR (fn (x y) (...))

fun_start_#lambda_0:
push rbp
mov rbp, rsp
sub rsp, 8*2
mov rax, [rbp - 8*-2]
add rax, [rbp - 8*-3]
mov rsp, rbp
pop rbp
ret
fun_finish_#lambda_0:

; 2. WRITE (arity, label) at [r11], [r11 + 8]

; 3. INCREMENT r11, and set/tag rax
```

QUIZ: Call using (Arity, Code Label)

What asm would we get for `<(inc 5)>`?

Lets *use* the tag + arity to check inc is a function that takes the given number of parameters.

```

; 1. compute/push params

; 2. LOAD `fn` from stack-slot into `rax`

; 3. CHECK `rax` has function tag 0101

; 4. STRIP tag

; 5. CHECK arity

; 6. CALL code-label

```

Now, instead of SEGFAULT-ing if we call a non-function, or use the wrong number of params, we abort appropriately.

Free Variables

What if the function uses variables that are **not** parameters?

```

(let (one 1)
  (let (inc (fn (x) (+ x one)))
    (inc 99)
  ))

```

Problem: "Free" Variables

The variable *one* is *free* in the body of *inc*.

- It is not a *parameter* of the function,
- It is not *let-bound* in the function.

Where do we get its value from?

Values

Idea 3: Function = (Arity, Code Label, Free Vars)

Let's **package** the values of the free variables with function.

Idea Extend our "vec-like" function representation to *package* the values of free variables together with the code label and arity.

arity	code-label	code-label	fv_1	...	fv_n
-------	------------	-----------------------	------	-----	------

- *arity*: the number of parameters, (offset 0)
- *code label*: the entry point of the function's code (offset 8),
- *number of*: free variables: in the body (offset 16).
- *values of*: free variables in the body (offset 24...).

Let's try to compile using this new representation

```
(let (one 1) ; 1. BIND `one`
  (let (inc ; 3. BIND `inc`
    ((fn (x) (+ x one)) ; 2. DEFINE `fn (x) ...`
      (inc 99)) ; 4. CALL `inc`
  ))
```

How will the DEFINE code change?

How will the CALL code change?

The assembly for the above will look like

```

mov qword ptr [rbp - 8*2], 2 ; <(let (one ...))>
; 1. DEFINE: <(fn (x) (+ x one))>
; → 1a. CODE for `fn (x) ...`
; → 1b. CODE for closure-vector
mov [rbp - 8*3], rax ; <(let (inc ...))>
; → 2. CALL <(inc 99)>

```

As before the DEFINE will have two parts

- the code for the *function body*, and
- the code to create the *closure-vector* on the heap.

QUIZ: Definition's Function Body

Fill in the code for the *function body* for `fn (x) (+ x one)`.

```

jmp fun_finish_#lambda_0
fun_start_#lambda_0:
; setup
push rbp
mov rbp, rsp
sub rsp, 8*3

; load free variables ... from where?

; <(+ x one)>

; teardown
mov rsp, rbp
pop rbp
ret
fun_finish_#lambda_0:

```

QUIZ: Definition's Closure Vector

Next, lets fill in the definition of the closure vector itself.

Recall, there are *four* parts

1. *Arity*
2. *Code-label*
3. *Number of free vars* (**how** do we know which ones?)
4. *Values of free vars* (**where** do these values live?)

```

; write arity
mov rcx, 1
mov [r11], rcx

; write label
lea rcx, [rel fun_start_#lambda_0]
mov [r11 + 8], rcx

; write free arity
mov rcx, 1
mov [r11 + 16], rcx

; write free vars
mov rcx, [rbp - 8*2]
mov [r11 + 24], rcx

; bump r11 and set/tag rax to closure address
mov rax, r11
add r11, 32
add rax, 5

```

QUIZ: Call using Closure

Fill in the assembly for the **call** (inc 99)

- What is the call-target?
- What are the params?

```

; push the args

; load closure pointer

; check tag & arity & strip tag

; get call-target

; push "closure" as first arg!

; call!

```

How to determine which variables are “free”?

```
fn free_vars_defn(defn: &Defn) -> Vec<String> {
  let expr = Defn(defn.clone());
  let vars = free(&expr);
  let mut vars = vars.into_iter().collect();
  vars.sort(); vars.dedup();
  vars
}
```

QUIZ: *Fill in the definition of fv*

```
fn free(e: &Expr) -> im::HashSet<String> {
  match e {
    Num(_) | True | False | Input | Nil =>
      _____

    Add1(e) | Sub1(e) | Neg(e)
    | Loop(e) | Break(e) | Print(e) | Get(e, _) =>
      _____

    Call1(x, e) | Set(x, e) =>
      _____

    If(e1, e2, e3) =>
      _____

    Bin(_, e1, e2) | Vec(e1, e2) =>
      _____

    Call2(f, e1, e2) =>
      _____

    Block(es) =>
      _____

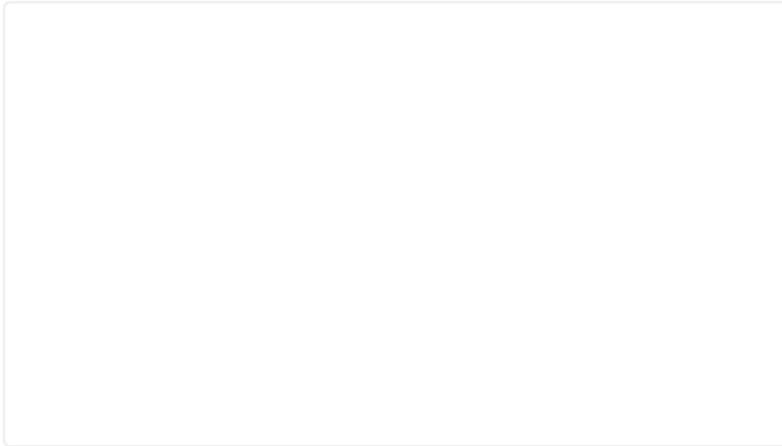
    Var(x) =>
      _____

    Let(x, e1, e2) =>
      _____

    Defn(defn) =>
      _____
      _____
  }
}
```

Recursion

What problems do *you* anticipate with recursive functions?



How might *you* solve them?

