Functional Programming Language Design

Christian J. Rudder

January 2025

Contents

This page is left intentionally blank.

Big thanks to **Professor Nathan Mull**
for teaching CS320: Concepts of Programming Languages
at Boston University [1].
Content in this document is based on content provided by Mull.

Disclaimer: These notes are my personal understanding and interpretation of the course material. They are not officially endorsed by the instructor or the university. Please use them as a supplementary resource and refer to the official course materials for accurate information.

— 1 —

The Interpretation Pipeline

1.0.1 Handling Lambda Recursion

We have Ω which allows us to do recursion, but we need self-referencing.

Definition 0.1: Fixed-point Combinator

A **fixed point** is a value unchanged by a transformation (e.g., the fixed point of f is some value x such that, $f(x) = x$). A fixed-point combinator is a higher-order function that satisfies:

$$\texttt{FIX}\ f = f(\texttt{FIX}f)$$

i.e., functions `FIX` and f when applied returns f whose argument is the original application. This enables recursion, as there is a self reference in scope. This unfolds to an infinite series of applications: $(\texttt{FIX}\ f = f(\texttt{FIX}f) = f(f(\texttt{FIX}f)) = f(f(f(\texttt{FIX}f))) = \ldots)$. Whether or not it converges depends on the behavior of f (i.e., a base-case).

Example 0.1: Writing Recursive Functions

Say we defined the following recursive factorial function, extending our lambda syntax:

$$\texttt{FACT} \triangleq \lambda n.\texttt{if}\ \ n = 0\ \ \texttt{then}\ \ 1\ \ \texttt{else}\ \ n * \texttt{FACT}(n-1)$$

To supply `FACT` with its own definition, we may preform an intermediary step:

$$\texttt{FACT'} \triangleq \lambda f.\lambda n.\texttt{if}\ \ n = 0\ \ \texttt{then}\ \ 1\ \ \texttt{else}\ \ n * (ff(n-1))$$

We define `FACT'`, which takes an additional function f to supply its recursive case. Now, we can apply `FACT'` to itself to render our desired `FACT` function:

$$\texttt{FACT} \triangleq\ \texttt{FACT'}\ \texttt{FACT'}$$

For example, let's supply 3 to `FACT`:

$$\begin{aligned}
\texttt{FACT}\ 3 &= (\texttt{FACT}'\ \texttt{FACT}')\ 3 && \text{Definition of FACT}\\
&= ((\lambda f.\,\lambda n.\,\texttt{if}\ \ n = 0\ \ \texttt{then}\ \ 1\ \ \texttt{else}\ \ n \times (f\ \ f\ \ (n-1)))\ \texttt{FACT}')\ 3 && \text{Definition of FACT'}\\
&\to (\lambda n.\,\texttt{if}\ \ n = 0\ \ \texttt{then}\ \ 1\ \ \texttt{else}\ \ n \times (\texttt{FACT}'\ \texttt{FACT}'\ (n-1)))\ 3 && \text{Application to FACT'}\\
&\to \texttt{if}\ \ 3 = 0\ \ \texttt{then}\ \ 1\ \ \texttt{else}\ \ 3 \times (\texttt{FACT}'\ \texttt{FACT}'\ (3-1)) && \text{Application to } n\\
&\to 3 \times (\texttt{FACT}'\ \texttt{FACT}'\ (3-1)) && \text{Evaluating } \texttt{if}\\
&\to \ldots\\
&\to 3 \times 2 \times 1 \times 1\\
&\to^* 6 && [2]
\end{aligned}$$

■

We make the following distinction to emphasize the meaning of a fixed-point:

Theorem 0.1: The identity function & fixed-points

Any function f is a fixed-point of the identity function I $(\lambda x.x)$, i.e., $I\ f = f$.

Our previous implementation of `FACT` in Example (0.1) was manual. This would be quite tedious for every recursive function. We can automate this with the following fixed-point combinator:

Definition 0.2: Y-Combinator

In lambda calculus, the **Y combinator** is a fixed-point combinator of form:

$$\mathtt{Y} \triangleq \lambda f.\, (\lambda x.\, f(x\ x))\, (\lambda x.\, f(x\ x))$$

E.g., $Y\ f = (\lambda x.\, f(x\ x))\, (\lambda x.\, f(x\ x)) = f((\lambda x.f(xx)(\lambda x.f(xx)))) = f(f(\ldots)) = \ldots.$

Example 0.2: Factorial with Y-Combinator

We can now define `FACT` using the Y combinator:

$$\begin{aligned} \mathtt{FACT} &\triangleq \lambda f.\lambda n.\mathtt{if}\ \ n = 0\ \ \mathtt{then}\ \ 1\ \ \mathtt{else}\ \ n * (f(n-1)) && (\text{ Definition of FACT' }) \\ \mathtt{Y\ FACT}\ 3 &= ((\lambda x.\, \mathtt{FACT}(x\ x))\, (\lambda x.\, \mathtt{FACT}(x\ x)))\ 3 && (\ [\mathtt{FACT}/f]\mathtt{Y}\ ) \\ &= \mathtt{FACT}\ ((\lambda x.\, \mathtt{FACT}(x\ x))\ (\lambda x.\, \mathtt{FACT}(x\ x)))\ 3 && (\ [\lambda x.\, \mathtt{FACT}(x\ x)/x]\mathtt{FACT}(x\ x)\ ) \end{aligned}$$

Remember that `FACT` still requires two arguments f and n, for which we now supply:

$$\begin{aligned} &= \mathtt{if}\ \ 3 = 0\ \ \mathtt{then}\ \ 1\ \ \mathtt{else}\ \ 3 * ((\lambda x.\, \mathtt{FACT}(x\ x))\ (\lambda x.\, \mathtt{FACT}(x\ x))(3-1)) \\ &= \mathtt{if}\ \ 3 = 0\ \ \mathtt{then}\ \ 1\ \ \mathtt{else}\ \ 3 * \mathtt{FACT}((\lambda x.\, \mathtt{FACT}(x\ x))\ (\lambda x.\, \mathtt{FACT}(x\ x)))\ (3-1) \\ &= \mathtt{if}\ \ 3 = 0\ \ \mathtt{then}\ \ 1\ \ \mathtt{else}\ \ 3 * (\mathtt{Y\ FACT}(3-1)) \\ &\vdots \\ &= \mathtt{if}\ \ 2 = 0\ \ \mathtt{then}\ \ 1\ \ \mathtt{else}\ \ 2 * (\mathtt{Y\ FACT}(2-1)) \\ &= \mathtt{if}\ \ 1 = 0\ \ \mathtt{then}\ \ 1\ \ \mathtt{else}\ \ 1 * (\mathtt{Y\ FACT}(1-1)) \\ &= \mathtt{if}\ \ 0 = 0\ \ \mathtt{then}\ \ 1\ \ \mathtt{else}\ \ 0 * (\mathtt{Y\ FACT}(0-1)) \end{aligned}$$

We hit the base-case and then evaluate $3 * (2 * (1 * 1))$ to get 6. ■

If we aren't careful step three of our derivation in Example (0.2) could lead to an infinite loop:

Definition 0.3: Strict vs. Lazy Evaluation

Strict evaluation means that all arguments to a function are evaluated before the function is applied, i.e., CBV (call-by-value).
Lazy evaluation means that an argument to a function is not evaluated until it is actually used in the body of the function, i.e., CBN (call-by-name).

Theorem 0.2: Y-Combinator & Lazy-Evaluation

The Y-combinator only works in lazy-evaluation settings. In strict evaluation, the Y-combinator will infinitely reduce.

To stop this we introduce another type of combinator for eager evaluation:

Definition 0.4: Z-Combinator

The Z-combinator is a fixed-point combinator that works in strict evaluation settings:

$$Z \triangleq \lambda f. \big(\lambda x. f\big(\lambda v. x\, x\, v\big)\big)\big(\lambda x. f\big(\lambda v. x\, x\, v\big)\big).$$

Example 0.3: Factorial with Z-Combinator (Part-1)

We now define the `FACT` using the Z combinator:

$$\begin{aligned} \texttt{Z FACT 3} &= ((\lambda x. \texttt{FACT}(\lambda v.x\ x\ v))\,(\lambda x. \texttt{FACT}(\lambda v.x\ x\ v)))\ 3 && (\ [\texttt{FACT}/f]\texttt{Z}\ ) \\ &= \texttt{FACT}(\lambda v.(\lambda x. \texttt{FACT}(\lambda v.x\ x\ v))\ (\lambda x. \texttt{FACT}(\lambda v.x\ x\ v)\ v))\ 3 && (\ \text{Application}\ ) \end{aligned}$$

The extra argument waiting for a value delays `Z` long enough to evaluate `FACT`:

$$\begin{aligned} &= \texttt{if}\ \ 3 = 0\ \ \texttt{then}\ \ 1\ \ \texttt{else}\ \ 3 * (\lambda v.(\lambda x. \texttt{FACT}(\lambda v.x\ x\ v))\ (\lambda x. \texttt{FACT}(\lambda v.x\ x\ v))\ v)\ (3-1))) \\ &= \texttt{if}\ \ 3 = 0\ \ \texttt{then}\ \ 1\ \ \texttt{else}\ \ 3 * (\lambda v.(\lambda x. \texttt{FACT}(\lambda v.x\ x\ v))\ (\lambda x. \texttt{FACT}(\lambda v.x\ x\ v))\ v)\ (2))) \\ &= \texttt{if}\ \ 3 = 0\ \ \texttt{then}\ \ 1\ \ \texttt{else}\ \ 3 * (\lambda x. \texttt{FACT}(\lambda v.x\ x\ v))\ (\lambda x. \texttt{FACT}(\lambda v.x\ x\ v))\ 2 \\ &= \texttt{if}\ \ 3 = 0\ \ \texttt{then}\ \ 1\ \ \texttt{else}\ \ 3 * \texttt{FACT}(\lambda v.(\lambda x. \texttt{FACT}(\lambda v.x\ x\ v))\ (\lambda x. \texttt{FACT}(\lambda v.x\ x\ v))\ v)\ \ 2 \\ &= \texttt{if}\ \ 3 = 0\ \ \texttt{then}\ \ 1\ \ \texttt{else}\ \ 3 * (\texttt{Z FACT}\ 2) \end{aligned}$$

This assumes that $\top$ (truthy) `if` expressions don't evaluate the `else` branch. ■

Last example touches on the idea of short-circuiting:

Definition 0.5: Short-Circuiting

Short-circuiting is a semantic trick which skips additional computation of a boolean expressions if some former part of the expression is sufficient to determine the value of the entire expression. For example ($\mathbb{B}$ is the set of booleans):

$$\frac{e_1 \Downarrow \bot}{e_1 \mathbin{\&\&} e_2 \Downarrow \bot}\ (\text{andEvalFalse}) \qquad \frac{e_1 \Downarrow \top \quad e_2 \Downarrow v,\ v \in \mathbb{B}}{e_1 \mathbin{\&\&} e_2 \Downarrow v}\ (\text{andEvalTrue})$$

$$\frac{e_1 \Downarrow \top}{e_1 \mid\mid e_2 \Downarrow \top}\ (\text{orEvalTrue}) \qquad \frac{e_1 \Downarrow \bot \quad e_2 \Downarrow v,\ v \in \mathbb{B}}{e_1 \mid\mid e_2 \Downarrow v}\ (\text{orEvalFalse})$$

$$\frac{e_1 \Downarrow \top \quad e_2 \Downarrow v}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Downarrow v}\ (\text{ifTrueEval}) \qquad \frac{e_1 \Downarrow \bot \quad e_3 \Downarrow v}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Downarrow v}\ (\text{ifFalseEval})$$

Notice that in (andEvalFalse), (orEvalTrue), and (ifTrueEval) the second expression is never evaluated. This is a form of **short-circuiting**.

1.0.2 Environments: Variable Binding Data-Structure

Though Y and Z combinators allow us to write recursive functions, this method quickly grows unwieldy as the complexity of our programs increases. Things like variable bindings and jumping between scopes become difficult to manage. That's where environments come in:

Definition 0.6: Environment

An **environment** is a data-structure that keeps track of **variable bindings**, i.e., associations between variables and their corresponding values. Environments are written as finite mappings:

$$\{x \mapsto v,\ y \mapsto w,\ z \mapsto f\}$$

Where each variable is mapped to a value. We may use such data-structure for state configurations. For example $\langle \{x \mapsto \lambda y.y\}, x\rangle \Downarrow v$. We shall denote environments as $\mathcal{E}$.

Theorem 0.3: Substitution vs. Environment Model

When we care about the speed of our program, the substitution model quickly becomes inefficient. This is because we have to read our entire program to find free variables and handle additional logic. Though we track state in configurations, the program itself is **still functionally pure**.

Here are the following operations we can preform on environments:

Definition 0.7: Operations on Environments

Environments support basic operations for managing variable bindings, similar to a map:

- $\varnothing$ — represents the empty environment (OCaml: `empty`).
- $\langle \mathcal{E} \rangle$ — represents the current environment (OCaml: `env`).
- $\langle \mathcal{E}[x \mapsto v] \rangle$ — adds a new binding of variable x to value v (OCaml: `add x v env`).
- $\langle \mathcal{E}(x) \rangle$ — looks up the value of variable x (OCaml: `find_opt x env`).
- $\langle \mathcal{E}(x) = \bot \rangle$ — indicates that x is unbound in the environment (OCaml: `find_opt x env = None`).

Additionally, if a new binding is added for a variable that already exists, the new binding **shadows** the old one:

$$\mathcal{E}[x \mapsto v][x \mapsto w] = \mathcal{E}[x \mapsto w]$$

This next piece is text specific:

Definition 0.8: Extended Lambda Calculus

Moving forward in the text, when we use **Lambda Calculus$^+$ (LC$^+$)**, we will be referring to the following grammar (which we may add to momentarily):

```
<expr> ::= <expr><expr>
         | let <var> = <expr> in <expr>
         | let rec <var> <var> = <expr> in <expr>
         | <val>
<var> ::= [a-zA-Z]
<val> ::= λ<var>.<expr> | <num>
```

Definition 0.9: Semantic Closures

A variable bindings under an environment creates a **closure**. There are two types of closures:

$$(\mathcal{E}, \lambda x.\, e) \quad \text{and} \quad (f, \mathcal{E}, \lambda x.\, e)$$

Unnamed and named closures respectively. The former captures the environment and function body. The latter includes the function name, allowing for safe self-referencing:

$$f \mapsto (f, \mathcal{E}, \lambda x.\, e)$$

Let's give **LC**$^+$ some semantics:

Definition 0.10: LC$^+$ Semantics

Values and variables

$$\frac{}{\langle \mathcal{E}, \lambda x.\, e\rangle \Downarrow (\mathcal{E}, \lambda x.\, e)} \qquad \frac{\langle \mathcal{E}, x\rangle \neq \bot}{\langle \mathcal{E}, x\rangle \Downarrow \mathcal{E}(x)} \qquad \frac{}{\langle \mathcal{E}, n\rangle \Downarrow n}$$

Let expressions

$$\frac{\langle \mathcal{E}, e_1\rangle \Downarrow v_1 \qquad \langle \mathcal{E}[\,x \mapsto v_1],\ e_2\rangle \Downarrow v_2}{\langle \mathcal{E},\ \text{let } x = e_1 \text{ in } e_2\rangle \Downarrow v_2} \qquad \frac{\langle \mathcal{E}[\,f \mapsto (\,f,\ \mathcal{E}',\ \lambda x.\, e_1)\,],\ e_2\rangle \Downarrow v_2}{\langle \mathcal{E},\ \text{let rec } f\, x = e_1 \text{ in } e_2\rangle \Downarrow v_2}$$

Application (unnamed closure)

$$\frac{\langle \mathcal{E}, e_1\rangle \Downarrow (\mathcal{E}', \lambda x.\, e) \qquad \langle \mathcal{E}, e_2\rangle \Downarrow v_2 \qquad \langle \mathcal{E}'[x \mapsto v_2],\, e\rangle \Downarrow v}{\langle \mathcal{E},\ e_1\, e_2\rangle \Downarrow v}$$

Application (named closure)

$$\frac{\langle \mathcal{E}, e_1\rangle \Downarrow (f,\ \mathcal{E}',\ \lambda x.\, e) \qquad \langle \mathcal{E}, e_2\rangle \Downarrow v_2 \qquad \langle \mathcal{E}'[\,f \mapsto (\,f,\ \mathcal{E}',\ \lambda x.\, e)\,][x \mapsto v_2]\, e\rangle \Downarrow v}{\langle \mathcal{E},\ e_1\, e_2\rangle \Downarrow v}$$

Example 0.4: Named Closure Derivation

Let $\alpha := (f \mapsto (f, \{x \mapsto 0\}, \lambda y.x))$, and β define the following premises:

- $\langle \{x \mapsto 1, \alpha\}, f\rangle \Downarrow (f, \{x \mapsto 0\}, \lambda y.x)$
- $\langle \{x \mapsto 1, \alpha\}, 2\rangle \Downarrow 2$
- $\langle \{x \mapsto 1, \alpha\}, [y \mapsto 2]x\rangle \Downarrow 0$

We derive the following:

$$\dfrac{\langle \{x \mapsto 0\}, \lambda y.x\rangle \Downarrow (\{x \mapsto 0\}, \lambda y.x) \qquad \dfrac{\langle \{x \mapsto 0, \alpha\}, 1\rangle \Downarrow 1 \qquad \dfrac{\beta}{\langle \{x \mapsto 1, \alpha\}, f\ 2\rangle \Downarrow 0}\ \text{(NC)}}{\langle \{x \mapsto 0, \alpha\}, \text{let } x = 1 \text{ in } f\ 2\rangle \Downarrow 0}\ \text{(L)}}{\langle \{x \mapsto 0\},\ \text{let } f = \lambda y.\, x \text{ in let } x = 1 \text{ in } f\ 2\rangle \Downarrow 0}\ \text{(L)}$$

Shorthanded rule names, NC:= Named Closure, L:= Let. ■

Bibliography

[1] Nathan Mull. Cs320: Concepts of programming languages. Lecture notes, Boston University, Spring Semester, 2025. Boston University, CS Department.

[2] Cornell University. Fixed-point combinators. Lecture Notes, 2020.