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.

— 1 —

The Interpretation Pipeline

1.1 Semantic Evaluation

1.1.1 Small-step Semantics

In our previous derivations, we've been doing **Big-step semantics**:

Definition 1.1: Big-Step Semantics

Big-step semantics describes how well-formed expressions evaluate to a final value, without detailing each intermediate step.

Notation: We write $e \Downarrow v$ to mean that the expression e evaluates to the value v.
Example:

$$\text{(sub 10 (add (add 1 2) (add 2 3)))} \Downarrow 2$$

Here, we now introduce **Small-step semantics**:

Definition 1.2: Small-Step Semantics

Small-step semantics describes how an well-formed expressions reduce one step at a time until a final value is reached. **Notation:** $e \to e'$, means that e reduces to e' in a single step. These semantics are housed in a **configuration** with some state-structure and program. We write:

$$\underbrace{\langle S, p \rangle}_{\text{configuration}} \longrightarrow \underbrace{\langle S', p' \rangle}_{\text{transformation.}}$$

Where S is the state of the program and p is the program. The rightarrow shows the **transformation** or **reduction** of the program. For purposes in FP there is no state, hence:

$$\langle \varnothing, p \rangle \longrightarrow \langle \varnothing, p' \rangle$$

Moving forward we <u>**shorthand** this to $p \to p'$</u> for brevity. Inference rules outline the semantics of the language:

$$\frac{e_1 \to e_1'}{e_1 + e_2 \longrightarrow e_1' + e_2} \text{ (reduce-left-first)}$$

This rule states that if the e_1 position is reducible ($e_1 \to e_1'$), reduce it first.

Let's try defining some small-step semantics for a toy-language:

Example 1.1: Defining Grammars in Small-Step Semantics

Say we have some grammar that can only add from 0 to 9:

```
    <expr> ::= "(" <op> <expr> <expr> ")"
             | <int>
    <op>   ::= add
    <int>  ::= 0-9
```

Let's assume our language reads from **left-to-right** and define the semantics of `add`:

- **Both arguments are expressions:**

$$\frac{e_1 \to e_1'}{(\text{add } e_1\ e_2) \to (\text{add } e_1'\ e_2)}\ (\text{add-left})$$

- **Left argument is an integer:**

$$\frac{n \text{ is an int} \qquad e_2 \to e_2'}{(\text{add } n\ e_2) \to (\text{add } n\ e_2')}\ (\text{add-right})$$

- **Both arguments are integers:**

$$\frac{n_1 \text{ and } n_2 \text{ are int} \qquad n_1 + n_2 = v \text{ (integer addition)}}{(\text{add } n_1\ n_2) \to v}\ (\text{add-ok})$$

When writing semantics (in this case **add**) think, "What are all the possible argument states?" If we have `(add 5 e2)`, we must figure out what e2 represents in order to add.

We can almost think of these terminal-symbols as **base cases**. Additionally, since we read left-to-right, we do not need a rule for `(add e1 n)`, where $e1$ is an expression and n is an integer. This scenario is already covered by (add-left).

■

Tip: States can represent data structures like stacks, making them ideal for modeling stack-oriented languages. For example (ϵ is the empty program):

$$\begin{aligned} & (\varnothing,\ \texttt{push 2; push 3; add}) \\ \to\ & (2 :: \varnothing,\ \texttt{push 3; add}) \\ \to\ & (3 :: 2 :: \varnothing,\ \texttt{add}) \\ \to\ & (5 :: \varnothing,\ \epsilon) \end{aligned}$$

Definition 1.3: Multi-Step Semantics

Multi-step semantics captures the idea of reducing a configuration through **zero or more single-step reductions**. We write $C \to^\star D$ to mean that configuration C reduces to configuration D in zero or more steps. This relation is defined inductively with two rules:

$$\frac{}{C \to^\star C}\ \text{(reflexivity)} \qquad\qquad \frac{C \to C' \quad C' \to^\star D}{C \to^\star D}\ \text{(transitivity)}$$

These rules formalize:

- Every configuration reduces to itself *(reflexivity)*
- Multi-step reductions can be extended by single-step reductions *(transitivity)*
- If there are multiple ways to reduce $C \to^\star D$, then the semantics are **ambiguous**.

Example 1.2: Proving Multi-step Reductions

We show (add (add 3 4) 5)$\to^\star$ 14 based off semantics defined in Example (1.1). We will do multiple rounds of single-step reductions to yield a result. At each round, read bottom up:

1. $$\cfrac{\cfrac{\cfrac{}{\text{add 3 4} \to 7}\ \text{(add-ok)}}{\text{add 5 (add 3 4)} \to \text{add 5 7}}\ \text{(add-right)}}{\text{(add (add 5 (add 3 4)) 2)} \to \text{(add (add 5 7) 2)}}\ \text{(add-left)}$$

2. $$\cfrac{\cfrac{}{\text{(add 5 7)} \to 12}\ \text{(add-ok)}}{\text{(add (add 5 7) 2)} \to \text{(add 12 2)}}\ \text{(add-left)}$$

3. $$\cfrac{}{\text{(add 12 2)} \to 14}\ \text{(add-ok)}$$

Thus, (add (add 3 4) 5) $\to^\star$ 14. Following rules, we match on the next outermost pattern. Our very first reduction matches with (add-left). In particular, e_1 := (add 5 (add 3 4)), and we see that that's our starting value the next layer up.

Moreover, the trailing 2 in (add (add 5 (add 3 4)) 2), is not evaluated until the very last step (3), as we read from left-to-right. Even though *we can see it*, our patterns do not.

■

Though there is a clear distinction between big-step and multi-step semantics:

Theorem 1.1: Multi-Step vs. Big-Step Semantics

Both Multi-step and Big-step semantics have the same outcome, but differ in meaning.

$$e \to^\star v \quad \approx \quad e \Downarrow v$$

Unlike big-step semantics, small-step semantics concerns all possible patterns, allowing us to choose how we reduce terms, and **in which order**. This eliminates possible ambiguity in in big-step pattern matching.

1.1.2 Lambda Calculus

We briefly touched on **Lambda Calculus** in a previous section when discussing the Anonymous Function Definition (**??**). Here we go more in-depth:

Definition 1.4: Lambda Calculus Syntax

Lambda calculus is a formal system for representing computations using only single-argument functions, avoiding the need for multiple parameters or state. Lambda calculus has three basic constructs:

- **Variables:** x, y, z, etc.
- **Abstraction:** $\lambda x.e$ (a function that takes an argument x and returns expression e).
- **Application:** $e_1\, e_2$ (applying function e_1 to argument e_2).

The OCaml translation:

$$\lambda x.e \quad \equiv \quad \texttt{fun}\ x\ \texttt{->}\ e$$

Replacing '`fun`' with 'λ', and '`->`' with '`.`'. In pure lambda calculus (no additional syntax), **the only value** is a lambda **abstraction**.

Definition 1.5: The Identity Function

The identity function is a function that returns its argument unchanged. Represented as: $\lambda x.x$. If we introduce numbers, $(\lambda x.x)\ 5 \to 5$ (application). In OCaml, this is represented as: `(fun x -> x) 5`.

Before moving on we address a few notational conventions:

Definition 1.6: Symbols $\triangleq$ vs. :=

The symbol `:=` is used to define a variable or expression. The symbol $\triangleq$ is used to state that two expressions are equal **by definition**. For example, we may write a paper which reuses some large specific configuration $\langle \{\dots\}, \dots \rangle$; Instead of writing it again and again, we assign one variable to represent such idea:

$$\Delta_{\Pi}^{\star} \triangleq \langle \{\dots\}, \dots \rangle$$

Now throughout our paper, $\Delta_{\Pi}^{\star}$ signals to the reader that we are using this configuration. As opposed to `:=`, where we might temporarily assign the variable a to some value multiple times over the course of a document.

Next we look at what happens when we apply the identity function to itself:

Definition 1.7: The Diverging Term Ω

Lambda Calculus is capable of recursion; The backbone of which is the **diverging term**, for which we define as Ω (Omega):

$$\Omega \triangleq (\lambda x.x\,x)(\lambda x.x\,x)$$

The inner function $\lambda x.x\,x$ is sometimes called the **mockingbird combinator**, as it applies its argument to itself:

$$M \triangleq \lambda x.x\,x$$

Thus, $\Omega = M\,M$ creates an infinite loop of self-application.

Example 1.3: Showing Ω Divergence

We can show that Ω diverges by repeated self-application:

$$\begin{aligned}\Omega &\triangleq (\lambda x.x\,x)(\lambda x.x\,x)\\ &\to (\lambda x.(\lambda x.x\,x)\ (\lambda x.x\,x))\\ &\to (\lambda x.(\lambda x.(\lambda x.x\,x)\,(\lambda x.x\,x)))\\ &\to (\lambda x.(\lambda x.(\lambda x.(\lambda x.x\,x)\,(\lambda x.x\,x))))\\ &\to \dots\end{aligned}$$

Hence, Ω diverges as it continues to apply itself indefinitely. ■

Application has a formal definition in lambda calculus:

Definition 1.8: Application & β-Reduction

A (beta) **β-reduction** is the process of applying a function as an argument in lambda calculus:

1. $$\frac{e_1 \to e_1'}{e_1\ e_2 \to e_1'\ e_2}\ \text{(beta-left)}$$
2. $$\frac{e_2 \to e_2'}{(\lambda x.e_1)\ e_2 \to (\lambda x.e_1)\ e_2'}\ \text{(beta-right)}$$
3. $$\frac{}{(\lambda x.e)\ (\lambda y.e') \to [(\lambda y.e')/x]e}\ \text{(beta-ok)}$$

Where (1) we reduce the left-hand side of the application, (2) we reduce the right-hand side of the application, and (3) we apply a function to another function by substitution.

Note: (3) is our base case where **only values** are allowed to be substituted. We take a different approach and not **eagerly** evaluate expressions.

1. $$\frac{e_1 \to e_1'}{e_1\ e_2 \to e_1'\ e_2}\ \text{(beta-left)}$$
2. $$\frac{}{(\lambda x.e)\ e' \to [e'/x]e}\ \text{(beta-ok)}$$

Note: (2) though we don't evaluate e' immediately, we **still** expect it to be a value when evaluating the full expression. As when we evaluate, we are expecting a value, **not some unknown variable**.

Example 1.4: Simple β-Reduction

Consider the following example of β-reduction:

$$(\lambda x.x + 1)\ 2 \quad \to \quad [2/x](x + 1) \quad \to \quad 2 + 1 \to 3$$

Here, we supply the function with 2, substitute 2 for x in the body of the function, and then evaluate the expression, yielding 3. ■

Though we must be wary of what we are substituting for:

Definition 1.9: α-Equivalence

Two expressions are (alpha) **α-equivalent** if they differ only variable names. This formalizes the **principle of name irrelevance**: renaming bound variables does not change the meaning of an expression.

$$\lambda x.\lambda y.x \ =_\alpha \ \lambda v.\lambda w.v$$

In OCaml-like syntax:

$$\texttt{let x = 2 in x} =_\alpha \texttt{let z = 2 in z}$$

Substitution should preserve α-equivalence. If $e_1 =_\alpha e_2$, then for any term v, we have:

$$[v/x]e_1 =_\alpha [v/x]e_2$$

To continue we make the following distinction:

Definition 1.10: Free & Bound Variables

In lambda calculus, a variable in an expression can be either **free** or **bound**:

- A variable is **bound** if it is defined by a λ abstraction in the expression.
  - **E.g.,** $(\lambda x.x + 1)$, the variable x is bound.
- A variable is **free** if it is not bound by any enclosing λ abstraction.
  - **E.g.,** $(\lambda x.y + 1)$, the variable y is free.

Formally, the set of free variables in an expression e, written $FV(e)$, is defined inductively as:

$$\begin{gathered} FV(x) = \{x\} \quad \textit{(Alone variable)} \\ FV(\lambda x.e) = FV(e) \setminus \{x\} \quad \textit{(Function body)} \\ FV(e_1\ e_2) = FV(e_1) \cup FV(e_2) \quad \textit{(Application)} \end{gathered}$$

In just a moment, we will define substitution in a way that preserves α-equivalence. The high-level idea is that we should **avoid** substituting variables that are **bound** within an expression.

Now we define the semantics of substitution:

Definition 1.11: Substitution Semantics

Substitution replaces **free** occurrences of a variable with another expression. The rules are defined inductively as follows, where in $[v/x]e$, v is assumed to be a **value** (abstraction):

$$(1) \qquad [v/y]x = \begin{cases} v & \text{if } x = y \\ x & \text{otherwise} \end{cases}$$

$$(2) \qquad [v/y](\lambda x.e) = \begin{cases} \lambda x.e & \text{if } x = y \\ \lambda z.[v/y]([z/x]e) & \text{if } x \in \mathrm{FV}(v), \text{ then make fresh } z \\ \lambda x.[v/y]e & \text{otherwise} \end{cases}$$

$$(3) \qquad [v/y](e_1\ e_2) = ([v/y]e_1)\ ([v/y]e_2)$$

Moreover on (2)'s middle condition. $FV(v)$ means free variables in v, so if $v := (\lambda w.y)$ then $FV(v) = \{y\}$. Then $[v/x](\lambda\ y.x)$ would be a major problem:

$$\lambda y.\lambda w.y \ \equiv \ \lambda y.y \ \neq_\alpha \ \lambda y.x$$

The middle condition avoids this by swapping in a **fresh variable** z that does not have any conflicts in the body. **E.g.,** $[v/x](\lambda y.x(\lambda z.y))$, $y \in FV(v)$, ignoring freshness of z:

$$(\lambda z.v(\lambda z.z)) \neq_\alpha (\lambda y.x(\lambda z.y))$$

The abstraction $(\lambda z.y)$ **captures** the subbed z in $(\lambda z.z)$. Now we pick some arbitrary **fresh** variable z, such that no captures occur:

$$(\lambda u.y(\lambda z.u)) =_\alpha (\lambda y.x(\lambda z.y))$$

Here we chose the variable u as it does not conflict with the rest of the expression.

.

Example 1.5: Multi-step β-Reductions

Consider the following derivations of $(\lambda f.\lambda x.fx)(\lambda y.y)$ using our substitution semantics:

1. $$\frac{}{(\lambda f.\lambda x.fx)(\lambda y.y) \to [(\lambda y.y)/f](\lambda x.fx) = (\lambda x.(\lambda y.y)x)}\ \text{(beta-ok)}$$

Yielding $(\lambda x.(\lambda y.y)x)$. We **cannot** evaluate inner-terms. I.e., let $w := (\lambda y.y)x$, then we have, $(\lambda x.w)$. Hence, there is no rule in Definition (1.8) that allows further reduction. ■

There are times where we can't evaluate a term

Definition 1.12: Stuck Terms

A **stuck term** is a well-formed expression that cannot be reduced, yet is not a value. Applying a non-function value to an argument often causes such issue:

$$((\lambda x.yx)(\lambda x.x)) \to y(\lambda x.x)$$

Here, the variable y is free, and thus, cannot proceed with application. Hence, we are "stuck" in the evaluation. We can avoid such scenarios via **typing systems** (discussed later).

There are two main evaluation strategies in program evaluation:

Definition 1.13: Call-by-Value vs. Call-by-Name

There are two main evaluation strategies:

- **Call-by-value (CBV)** evaluates the argument *before* substitution.
- **Call-by-name (CBN)** substitutes the argument expression *directly* without evaluating it first.

We may illustrate this with the following rules:

$$(\text{CBV})\ \frac{e_1 \Downarrow \lambda x.e_1' \quad e_2 \Downarrow v_2 \quad [v_2/x]e_1' \Downarrow v}{e_1\ e_2 \Downarrow v} \qquad (\text{CBN})\ \frac{e_1 \Downarrow \lambda x.e_1' \quad [e_2/x]e_1' \Downarrow v}{e_1\ e_2 \Downarrow v}$$

The benefit of CBV is that it **only evaluates an argument once** and is reused. With CBN, the argument is evaluated **every time** it needs to be computed. This is good if an expensive computation is passed around, but barely touched in execution (e.g., expensive edge-case).

We saw this before in Definition (1.8). The first semantics were CBV, while the latter was CBN.

Tip: There are many evaluation strategies optimizing different aspects of computation. In addition to **CBV** and **CBN** there are: **Call-by-need** (lazy eval)—like call-by-name, but avoids recomputation by memoizing results. Used in Haskell. **Call-by-reference**—used in languages with pointers (functions receive variable references). **Call-by-sharing**—also pointer focused langues (functions receive object references).

Definition 1.14: Well-Scopedness and Closedness

We expand on the idea of **scope** in respect to free and bound variables:

- An expression e is **closed** if it contains **no free variables**, also called a **combinator**.
- An expression e is **well-scoped** if the expression is closed or is bound under the configuration's state.

Every closed expression is well-scoped, **but** not every well-scoped expression is closed.

Note: The expression $x \mapsto y$ means that x is mapped to y. Hence, x is bound to y.

Example 1.6: Closed vs. Open Terms

Recall that abstractions bind to their argument variable:

- **Open Term:** $(\lambda x.y)$ is *not closed*, since y is free.
- **Closed Term & Well-scoped:** $(\lambda x.\lambda y.y)$ is *closed*, since both x and y are bound.
- **Well-scoped:** $\langle \{x \mapsto (\lambda y.y)\}, (\lambda w.x) \rangle$ is *well-scoped*, since x is bound in state, but not closed.

■

Definition 1.15: Lexical vs Dynamic Scope

A variable's **scope** determines where in the program the variable can be referenced.

- **Lexical (or static) scope:** Textual delimiters define the scope of a binding.
- **Dynamic scope:** Bindings are determined at runtime based on the call stack. I.e., the most recent binding in the call stack is used regardless was declared.

To understand the difference between lexical and dynamic scoping:

Example 1.7: Dynamic vs Lexical Scoping

Consider the following Bash code:

```bash
f() { x=23; g; }
g() {  y=$x; }
f
echo $y    # prints 23
```

In Bash, the variable `x` is not defined in `g`, but since `f` called `g` and `x` was defined in `f`, `g` sees it. This is **dynamic scoping**. In contrast, consider the following Python code:

```python
x = 0
def f():
    x = 1
    return x

assert f() == 1
assert x == 0
```

Now consider the following OCaml code:

```ocaml
let x = 0
let f () =
    let x = 1 in
    x

let _ = assert (f () = 1)
let _ = assert (x = 0)
```

Both Python and OCaml use **lexical scoping**, meaning each use of `x` refers to the closest enclosing definition in the source code, not the caller's environment. ■

Theorem 1.2: Lambda Calculus – True-False Conditions

True,false, and if conditions are defined as:

$$\texttt{true} \triangleq \lambda x.\lambda y.x \qquad \texttt{false} \triangleq \lambda x.\lambda y.y$$

$$\texttt{IF} \triangleq \lambda cond.\lambda then.\lambda else.cond\ then\ else$$

E.g.,
(`IF true 1 2`) $\rightarrow [2/y][1/x](\lambda x.\lambda y.x) \rightarrow 1$.
(`IF false 1 2`) $\rightarrow [2/y][1/x](\lambda x.\lambda y.y) \rightarrow 2$.

1.1.3 Handling Lambda Recursion

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

Definition 1.16: 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 1.8: 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:

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

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 1.3: 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 (1.8) was manual. This would be quite tedious for every recursive function. We can automate this with the following fixed-point combinator:

Definition 1.17: 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., $Y\ f = (\lambda x.\, f(x\ x))\, (\lambda x.\, f(x\ x)) = f((\lambda x.f(xx)(\lambda x.f(xx)))) = f(f(\dots)) = \dots.$

Example 1.9: 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 (1.9) could lead to an infinite loop:

Definition 1.18: 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 1.4: 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 1.19: 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 1.10: 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 1.20: 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):

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

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

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

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

1.1.4 Environments: Variable Binding Data-Structure

Though as programs and systems grow more complex it may be difficult to keep track of variables. Say we jump to another function, or create a new thread. We need some way to keep track of the variable mappings. That's where environments come into play:

Definition 1.21: 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 (e.g., $\langle \{x \mapsto \lambda y.y\}, x\rangle$). We shall denote environments as $\mathcal{E}$.

Definition 1.22: Operations on Environments

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

- $\varnothing$ — represents the empty environment (OCaml: `empty`).
- $\mathcal{E}$ — represents the current environment (OCaml: `env`).
- $\mathcal{E}[x \mapsto v]$ — adds a new binding of variable x to value v (OCaml: `add x v env`).
- $\mathcal{E}(x)$ — looks up the value of variable x (OCaml: `find_opt x env`).
- $\mathcal{E}(x) = \bot$ — 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:

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

We've seen this before in Definition (1.14).

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.