Functional Programming Language Design

Christian J. Rudder

January 2025

Contents

Contents **1**

1 The Interpretation Pipeline **4**
1.1 Semantic Evaluation 5
1.1.1 Small-step Semantics 5
1.1.2 Lambda Calculus 8

Bibliography **18**

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

1.1 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 a complete expression evaluates directly to a final value, without detailing each intermediate step. It relates an expression to its result in a single derivation.

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 expression is reduced one step at a time. Each step transforms the current expression into a simpler one until no further reductions are possible.
Notation: We write $e \to e'$ to mean that e reduces to e' in a single step. The notation:

$$\underbrace{(S, p)}_{\text{configuration}} \longrightarrow \underbrace{(S', p')}_{\text{transformation.}}$$

Where S is the state of the program and p is the program. The rightarrow shows the **transformation** or **reduction** of the program. Since for our purposes OCaml *doesn't* have state, so we'd typically write:

$$(\varnothing, p) \longrightarrow (\varnothing, p')$$

Hence, moving forward we shorthand this to $p \to p'$ for brevity. We may describe the semantics for grammars in terms of small-step semantics using inference rules:

$$\frac{e_1 \to e_1'}{e_1 + e_2 \longrightarrow e_1' + e_2} \text{ (reduction)}$$

Where e is a well-formed expression that can be reduced to e', hence our premise "$e \to e'$".

We can use these small-step semantics to define evalutions in our grammar:

Example 1.1: Defining Grammars in Small-Step Semantics

Say we have part of some toy-language grammar:

```
<expr> ::= ( <op> <expr> <expr> )
         | <int>
<op>   ::= add | sub | eq
<int>  ::= ...
```

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

- **Both arguments are expressions:**

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

- **Left argument is an integer:**

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

- **Both arguments are integers:**

$$\frac{n_1 \text{ and } n_2 \text{ are integer literals}}{(\text{add } n_1\ n_2) \to n_1 + n_2}\ \text{(add-ok)}$$

The intuition is to think about our grammar, in this case **add**, and think, "What are all the possible argument states of add?" If we have `(add <expr> <expr>)`, we have to reduce `<expr>` before we can evaluate it. In cases like `(add 1 2)`, there is nothing left to reduce.

We can almost think of these terminal-symbols as **base cases**. Additionally, since we read left to right, (add <expr> 2) is impossible, as we should have evaluated the left-hand side first.

■

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:

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

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: Multi-step Reduction

We show (add (add 3 4) 5)$\to^\star$ 14 based off the semantics we defined in Example (1.1). We will do multiple rounds of single-step reductions to yield a final value:

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

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

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

Thus, (add (add 3 4) 5) $\to^\star$ 14. When deriving, we think like a compiler, and grab the next recursive call to reduce. Notice how our very first reduction matches with (add-left). In particular, $e_1 :=$ (add 5 (add 3 4)), and we see 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*, the computer does not.

■

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

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

Multi-step semantics bridges small-step and big-step semantics:

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

Where a well-formed expression e reduces to a value v via zero or more single-step reductions. This is approximately what big-step semantics aims to accomplish without underlying intermediate steps.

Unlike big-step semantics, small-step semantics allows us to choose how we reduce our terms in every step, and **in which order**. This eliminates possible ambiguity in the grammar.

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).

In particular:

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

Where we replace the '`fun`' with 'λ', and '`->`' with '`.`'.

Definition 1.5: The Identity Function

The identity function is a function that returns its argument unchanged. In lambda calculus, it is represented as: $\lambda x.x$. In particular, $(\lambda x.x)\ 5 \to 5$ (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 $(\{\dots\}, \dots)$; Instead of writing it again and again, we assign one variable to represent such idea:

$$\Delta^{\star}_{\Pi} \triangleq (\{\dots\}, \dots)$$

Now throughout our paper, $\Delta^{\star}_{\Pi}$ 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

The identity function, that we'll denote as I, when applied to itself is called the **diverging term**, for which we define as Ω:

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

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 applying it to itself:

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

This shows that Ω diverges as it continues to apply itself indefinitely. ■

Application has a formal definition in lambda calculus:

Definition 1.8: Application & β-Reduction

β-reduction is the process of applying a function to an argument in lambda calculus. We proceed with the small-step semantics for the application of two functions:

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. $$\frac{}{(\lambda x.e)\ (\lambda y.e') \to [(\lambda y.e')/x]e}\ \text{(beta-ok)}$$
4. For e.g., $e := x + x$ then,
$$[(\lambda x.e')/x]e = (\lambda x.e') + (\lambda x.e')$$

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:** (4) that the outer λ is discarded upon substitution, only the substituted body remains.

We can make this more compact and generalize to any expression e':

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, e' only needs to be a well-formed expression for a β-reduction.

Example 1.4: Simple β-Reduction

Consider the following example of β-reduction:

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

Here, we apply the function to the argument 2, substitute 2 for x in the body of the function, and finally evaluate the expression to get 3. ■

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

Definition 1.9: α-Equivalence

Two lambda calculus expressions are said to be **α-equivalent** (alpha) if they differ only by the names of their bound variables. 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:

```
let x = 2 in x + 1 =α let z = 2 in z + 1
```

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

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

$$\begin{aligned} FV(x) &= \{x\} \\ FV(\lambda x.e) &= FV(e) \setminus \{x\} \\ FV(e_1 \ e_2) &= FV(e_1) \cup FV(e_2) \end{aligned}$$

A variable is **bound** if it is not free.

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 recursively as follows:

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

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

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

Moreover, we pay close attention to (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 as,

$$\lambda y.\lambda w.y = \lambda y.y \neq_\alpha \lambda y.x,$$

is not α-equivalent. The condition accounts for this by making a **fresh variable** z that does not have any conflicts in the body. For example $[y/x](\lambda y.x(\lambda z.y))$ ignoring freshes:

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

Now we pick some arbitrary **fresh** variable z: argument with z:

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

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

2. $$\frac{\dfrac{}{(\lambda y.y)\,x \to [x/y](y) = x}\ \text{(beta-ok)}}{(\lambda x.(\lambda y.y)\,x) \to (\lambda x.x)}\ \text{(beta-left)}$$

Hence, $(\lambda f.\lambda x.fx)(\lambda y.y) \rightarrow^\star \lambda x.x.$ ■

Though we can't cover everything with our grammar:

Definition 1.12: Stuck Terms

A **stuck term** is a well-formed expression in lambda calculus that cannot be reduced, yet is not a value (i.e., not a lambda abstraction). 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 not bound to a function, so we cannot proceed with application. Since $y(\lambda x.x)$ is not a lambda and cannot reduce, it is stuck. We can avoid such scenarios via **typing systems**.

There are two main evaluation strategies in lambda calculus:

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

Call-by-value (CBV) and **Call-by-name (CBN)** are two evaluation strategies in lambda calculus and functional programming.

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

We may illustrate this with the following rules:

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

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 the execution.

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

Lambda Calculus redefines **scope** in terms of **free** and **bound** variables:

- An expression e is **well-scoped** if every **free variable** in e is bound somewhere in the surrounding context.
- An expression e is **closed** if it contains **no free variables**. That is, all variables in e are bound within e itself.

Every closed expression is well-scoped by definition, but not every well-scoped expression is closed. Closed terms are especially important because they are self-contained and can be evaluated without needing an external context.

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:** $(\lambda x.\lambda y.y)$ is *closed*, since both x and y are bound.

■

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** refers to the textual delimiters to 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 of where the function was defined.

Most modern programming languages use lexical scoping because it makes code easier to understand and reason about just by reading the source.

To understand the difference between lexical and dynamic scoping:

Example 1.7: Dynamic vs Lexical Scoping

Consider the following Bash code:

```
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:

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

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

Now consider the following OCaml code:

```
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. ■

Definition 1.16: 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, such as a number, function, or expression.

Definition 1.17: 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:

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

Bibliography

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