Functional Programming Language Design

Christian J. Rudder

January 2025

Contents

Contents 1

1 The Interpretation Pipeline 4
1.1 Semantic Evaluation . . . 4
1.1.1 Small-step Semantics . . . 4

Bibliography 7

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:

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

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:

$$\frac{e1 \to e1'}{e1 + e2 \longrightarrow e1' + e2}\ \text{(reduction)}$$

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):

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

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 $C \to^\star D$, we say the small-step semantics is **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 reductions to yield a final value:

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

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

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

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, $e1$:= (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 7) 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.

■

Bibliography

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