

Theory of Programming Languages

NPRG085

Vít Šefl & Tomáš Petříček

Charles University

2026

Operational Semantics

Recall that operational semantics explains the meaning of a program by describing its evaluation as a sequence of computational steps. This is in contrast to other techniques, such as denotational semantics, which maps programs to mathematical objects that already have some well-established meaning.

Last week, you saw an example of *small-step* (or structural) operational semantics.

Operational Semantics

$$E ::= \mathbb{N} \mid E + E$$

$$V ::= \mathbb{N}$$

We have expressions E and values V . Small-step semantics works by defining a relation $\rightarrow \subseteq E \times E$:

$$\frac{V_1 + V_2 = V}{V_1 + V_2 \rightarrow V} \quad \frac{E_1 \rightarrow E'_1}{E_1 + E_2 \rightarrow E'_1 + E_2} \quad \frac{E_2 \rightarrow E'_2}{V_1 + E_2 \rightarrow V_1 + E'_2}$$

Notice: *reduction* rules vs *congruence* rules.

Operational Semantics

Our desired semantics is given by the reflexive transitive closure of $\rightarrow$ (denoted $\rightarrow^*$).

In some cases, you can find operational semantics that do not split the evaluation into these tiny steps and describe how to get the values directly. This is known as big-step (natural) operational semantics.

Operational Semantics

We define the relation $\Downarrow \subseteq E \times V$ (notice the values).

$$\frac{}{V \Downarrow V}$$
$$\frac{E_1 \Downarrow V_1 \quad E_2 \Downarrow V_2 \quad V = V_1 + V_2}{E_1 + E_2 \Downarrow V}$$

Because we directly get the final value, the process is simpler but we lose some finer details (e.g., order of evaluation). In general, big-step semantics cannot talk about programs that diverge!

Operational Semantics

We can add stores to natural semantics, as we did with structural semantics. For example, if the above language had variables (but not assignment):

$$\mathbf{Config} = E \times (\mathbf{Name} \rightarrow V)$$

We can then define $\Downarrow \subseteq \mathbf{Config} \times V$:

$$\frac{x \in \text{dom}(\sigma)}{\langle x, \sigma \rangle \Downarrow \sigma(x)}$$

Other rules are changed accordingly.

Lambda Calculus

Lambda calculus is a very simple functional language. It was first described by Alonzo Church (of the Church-Turing thesis fame) in the 1930s.

Lambda calculus consists of only variables and functions. Functions can be defined with the λ operator (hence the name) and applied to other expressions. We can summarize the expressions of LC as follows:

$$E ::= x \mid E E \mid \lambda x. E$$

Notice that the function application is a simple juxtaposition! $\lambda x.$ is called an *abstraction*.

Lambda Calculus

The application is just a function call, and abstraction is a (anonymous) function definition. By the way, this is why many programming languages use the name *lambda* to refer to their anonymous functions.

Some useful conventions:

$\lambda x.\lambda y.\lambda z.E$ is abbreviated as $\lambda xyz.E$

$E_1 E_2 E_3$ means $(E_1 E_2) E_3$ (i.e., left-associative)

$\lambda x.E_1 \dots E_n$ means $\lambda x.(E_1 \dots E_n)$ (i.e., λ extends right)

Variables

The abstraction λ binds a variable; it is a so-called *binder*. You have seen binders in formal systems before (e.g., $\forall$ in formal logic).

Each occurrence of a variable is either *bound* (if there is a corresponding binder that binds it) or *free*. In the expression $\lambda x.E$, all occurrences of x within E are bound.

The choice of a name for a bound variable should not matter. Renaming a bound variable is sometimes called α -conversion. If two expressions are equivalent up to renaming, we say they are α -equivalent.

Variables

The free variables of a lambda term can be defined recursively:

$$\mathbf{FV}(x) = \{x\}$$

$$\mathbf{FV}(E_1 E_2) = \mathbf{FV}(E_1) \cup \mathbf{FV}(E_2)$$

$$\mathbf{FV}(\lambda x.E) = \mathbf{FV}(E) \setminus \{x\}$$

Notice that a variable can be both free and bound in a single expression:

$$\lambda x.(\lambda y.xy)y$$

We generally avoid this confusion by renaming the variables.

Substitution

Free variables can be substituted for, replacing them with another expression. We will write $[x := E]$ for a substitution that replaces x with the expression E . A substitution can be applied to an entire term, in which case it applies to all free occurrences of the given variable:

$$x(\lambda y.yx)[x := \lambda z.z] = (\lambda z.z)(\lambda y.y(\lambda z.z))$$

However, there is a potential problem:

$$(\lambda y.x)[x := y] \neq \lambda y.y$$

We wanted to replace x with a free y , and in the resulting expression it became bound!

Substitution

A variable capture occurs when a free variable accidentally becomes bound. The fix is simple: we rename the bound variables if a capture would occur, giving us *capture-avoiding substitution*.

$$(\lambda y.x)[x := y] = (\lambda z.x)[x := y] = \lambda z.y$$

Formally:

$$x[x := S] = S$$

$$y[x := S] = y$$

$$(\lambda x.E)[x := S] = \lambda x.E$$

$$(\lambda y.E)[x := S] = \lambda y.E[x := S] \quad (\text{if } y \notin \mathbf{FV}(S))$$

$$(\lambda y.E)[x := S] = \lambda z.E[y := z][x := S] \quad (\text{for some fresh } z)$$

$$(E_1 E_2)[x := S] = (E_1[x := S])(E_2[x := S])$$

Semantics

Substitution allows us to define the one and only reduction rule in lambda calculus:

$$\overline{(\lambda x. E_1) E_2 \rightarrow E_1[x := E_2]}$$

This rule is known as β -reduction. A subexpression that is β -reducible is called a β -redex. The congruence rules have the following form:

$$\frac{E \rightarrow E'}{\lambda x. E \rightarrow \lambda x. E'} \quad \frac{E_1 \rightarrow E'_1}{E_1 E_2 \rightarrow E'_1 E_2} \quad \frac{E_2 \rightarrow E'_2}{E_1 E_2 \rightarrow E_1 E'_2}$$

Semantics

Notice that these rules are not deterministic. You might have an expression that admits multiple different reductions!

$$y((\lambda x.x)z) \leftarrow ((\lambda x.x)y)((\lambda x.x)z) \rightarrow ((\lambda x.x)y)z$$

Thankfully, in this case it does not matter *too* much since the reduction sort of meets up in the middle:

$$y((\lambda x.x)z) \rightarrow yz \leftarrow ((\lambda x.x)y)z$$

Is that always the case?

Confluence

Definition: A reduction system is *confluent* if, whenever $E \rightarrow^* E_1$ and $E \rightarrow^* E_2$, there always exists E' such that $E_1 \rightarrow^* E'$ and $E_2 \rightarrow^* E'$.

That is to say, if we pick different redexes, we will eventually arrive at the same result.

As it turns out, the lambda calculus is confluent. We will skip the proof here, but feel free to check it in a textbook.

Confluence

Because of confluence, it makes sense to talk about the *normal form* even when the reduction is not deterministic. An expression is in normal form if it has no β -redexes.

Not every expression has a normal form! Consider

$$\underline{(\lambda x.xx)(\lambda x.xx)} \rightarrow \underline{(\lambda x.xx)(\lambda x.xx)} \rightarrow \dots$$

However, if the normal form exists, it is unique (up to α -conversion). Suppose that N_1 and N_2 are two normal forms of N . By confluence, there exists N' such that $N_1 \rightarrow^* N'$ and $N_2 \rightarrow^* N'$. If $N_1 \neq N'$ then N_1 could not have been in a normal form (similarly for N_2), so it must be the case that $N_1 = N' = N_2$.

Values

What about values? Previously, we defined values as a new syntactic category ($V ::= \dots$) and used it to enforce the evaluation order. We can define a syntactic category for lambda terms in a normal form:

$$V ::= \lambda x_1 \dots x_n. y \ V_1 \dots V_m \quad (n, m \geq 0)$$

And while it is technically possible to use it to enforce evaluation order, e.g call-by-value, it is easier to define separately.

$$\frac{}{(\lambda x. E) V \rightarrow E[x := V]}$$
$$\frac{E_1 \rightarrow E'_1}{E_1 \ E_2 \rightarrow E'_1 \ E_2} \quad \frac{E \rightarrow E'}{V \ E \rightarrow V \ E'}$$

Reduction Strategy

A reduction strategy tells us which redex to reduce first. If we do not allow reduction to occur inside functions, we have:

- ▶ *Call by value* Functions can only be applied to values. That is, reduce $(\lambda x.E_1)E_2$ only when E_2 is fully evaluated.
- ▶ *Call by name* Apply functions as soon as possible, even if the arguments are not yet evaluated.

Reduction Strategy

Suppose that we have a variant of the lambda calculus with integers and addition. We can see the difference between the two strategies in the following example:

$$\begin{aligned}(\lambda x.x + x)(\underline{1 + 1}) &\rightarrow \underline{(\lambda x.x + x) 2} \rightarrow \underline{2 + 2} \rightarrow 4 \\ \underline{(\lambda x.x + x)(1 + 1)} &\rightarrow \underline{(1 + 1) + (1 + 1)} \\ &\rightarrow 2 + \underline{(1 + 1)} \rightarrow \underline{2 + 2} \rightarrow 4\end{aligned}$$

Reduction Strategy

We can also allow reduction to occur inside functions (this might be necessary if you want to find the normal form as defined above). We will discuss these strategies later.

Although the normal form is unique, whether we find it depends on the actual strategy. Consider

$$\begin{aligned}(\lambda xy.x) z \ ((\lambda w.ww)(\lambda w.ww)) &\rightarrow \underline{(\lambda xy.x) z \ ((\lambda w.ww)(\lambda w.ww))} \\&\rightarrow \underline{(\lambda y.z) \ ((\lambda w.ww)(\lambda w.ww))} \\&\rightarrow z\end{aligned}$$

Currying

In lambda calculus, all functions are unary. We can simulate a function with multiple parameters using multiple abstractions.

$$(\lambda x.(\lambda y.x + y))\ 1\ 2 \rightarrow (\lambda y.1 + y)\ 2 \rightarrow 1 + 2 \rightarrow 3$$

This concept is known as *currying*!

Recursion

Lambda calculus has no concept of a definition, so one might expect recursion to be impossible. Let us take a look at a standard factorial function.

$$f := \lambda n. \mathbf{if} \ (n = 0) \ 1 \ (n \cdot f \ (n - 1))$$

We can take the recursive occurrence of f and turn it into a parameter without changing the meaning:

$$f := (\lambda g \ n. \mathbf{if} \ (n = 0) \ 1 \ (n \cdot g \ (n - 1))) \ f$$

Recursion

Notice that our definition now has the form:

$$f := G f$$

Recall that any x that satisfies $x = h(x)$ is a fixed point of h . In our case, f is a fixed point of the function G . If we can find these fixed points, we can express any recursion!

Define Y as:

$$Y := \lambda f. (\lambda x. f (x x)) (\lambda x. f (x x))$$

Recursion

With some equational reasoning, we can show that:

$$\begin{aligned} Y\ G &= (\lambda x. G\ (x\ x))(\lambda x. G\ (x\ x)) \\ &= G\ ((\lambda x. G\ (x\ x))(\lambda x. G\ (x\ x))) \\ &= G\ (Y\ G) \end{aligned}$$

That is, $Y\ G$ is precisely the fixed point f we were looking for!
Indeed, $Y\ G\ n$ computes the factorial of n .

Y is a so-called *fixed-point combinator*.

Turing Completeness

The previous result seems to indicate that lambda calculus is computationally powerful. Indeed, as we will see later, unbounded recursion allows us to simulate a Turing machine, which makes lambda calculus Turing complete.

However, how do you simulate a state machine and an infinite tape if all you have are functions? We could of course extend lambda calculus with some new data types, but in a couple of weeks, I will show you that functions are perfectly sufficient!

In fact, we can encode all useful data types using only functions. However, whether we want to do so in practice is a different question.

Type Systems

Type systems exist to provide some level of guarantees about the program. Typically, programming languages use these systems to show that the program does not have some unwanted behavior (e.g., adding a number and a string together).

In particular, later in this course, we will add types to our language(s) and establish that the type system plays nicely with reduction: reduction preserves types, and typed programs do not get stuck!

Type Systems

In general, a type system is a logical system that assigns types to expressions (statements, definitions, etc) of a given language. At its core is the relation $E : T$ (E has the type T). We can use a type system to answer the following:

- ▶ Check that $E : T$. (*type checking*)
- ▶ Find T such that $E : T$. (*type inference*)
- ▶ Find E such that $E : T$. (*type inhabitation*)

Every type system is capable of type checking; some are capable of type inference. Type inhabitation is only a consideration to special languages such as proof assistants.

Type Systems

As an example, here is a simple type system for an expression language with numbers, booleans, addition, and if-then-else:

$$\frac{n \in \mathbb{Z}}{n : \mathbf{Num}} \quad \frac{}{\mathbf{true} : \mathbf{Bool}} \quad \frac{}{\mathbf{false} : \mathbf{Bool}}$$
$$\frac{E_1 : \mathbf{Num} \quad E_2 : \mathbf{Num}}{E_1 + E_2 : \mathbf{Num}} \quad \frac{C : \mathbf{Bool} \quad E_1 : T \quad E_2 : T}{\mathbf{if } C E_1 E_2 : T}$$

Type Systems

What happens if we add variables? What is the type of $x + 2$?

We need to keep track of which variables have what type. This is done with a *typing context*, typically written as Greek uppercase Γ . A typing context is a set of pairs, each containing a variable and its type.

Type Systems

Our typing relation then becomes $\Gamma \vdash E : T$ (in the context Γ , E has the type T). The previous rules need to be modified to include this context, but otherwise remain the same. We do need an additional rule for the variable itself:

$$\frac{(x, T) \in \Gamma}{\Gamma \vdash x : T}$$

With that, we can easily show that $\{(x, \mathbf{Num})\} \vdash x + 2 : \mathbf{Num}$.