Chapter 15

Abstract Machine Semantics

Introduction

In these notes we will be talking about *control* structures in programming languages. To do so, we'll introduce yet another style of *operational semantics*, one that makes program control more apparent. We have already discussed *big-step semantics*, which take the form $t \Downarrow v$, and we've discussed two kinds of *small-step semantics*: *structural operational semantics* and *reduction semantics*. We'll now talk about a third kind of small-step semantics, called *abstract machine semantics*. Abstract machine semantics operationalize the intuition that we discussed regarding how to think about running a reduction semantic: given a program, you have to somehow find a way to decompose the program into an evaluation context and redex, then reduce the redex, and then plug the result back into the context, and then repeat these steps until a final outcome results. Abstract machines typically inject more systematic method into this process. As such, this semantic approach is a useful guide toward implementing compilers, which must translate your high-level language into low-level instructions that run on some machine with a bunch of registers, some stack memory, and some heap memory.[1]

A Simple Language

Our running example for this introduction is, as usual, a language of Boolean expressions. The language is bare-bones, but has enough content to capture most of the concepts that concern us at this time. We'll start with the language reduction semantics, and then will return to present it as an abstract machine.

Syntax

$$t \in \textsc{Term}, \quad v \in \textsc{Val}, \quad r \in \textsc{Redex}, \quad E \in \textsc{ECtxt}$$

$$\begin{array}{rcl}
t & ::= & \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\ t\ \mathsf{then}\ t\ \mathsf{else}\ t \mid t\ \mathsf{and}\ t \mid t\ \mathsf{or}\ t \mid t\ \mathsf{xor}\ t \mid \mathsf{not}\ t \\
v & ::= & \mathsf{true} \mid \mathsf{false} \\
r & ::= & \mathsf{if}\ v\ \mathsf{then}\ t\ \mathsf{else}\ t \mid v\ \mathsf{and}\ t \mid v\ \mathsf{or}\ t \mid v\ \mathsf{xor}\ v \mid \mathsf{not}\ v \\
E & ::= & [\,] \mid E[\mathsf{if}\ \square\ \mathsf{then}\ t\ \mathsf{else}\ t] \mid E[\square\ \mathsf{and}\ t] \mid E[\square\ \mathsf{or}\ t] \mid E[\square\ \mathsf{xor}\ t] \mid E[v\ \mathsf{xor}\ \square] \mid E[\mathsf{not}\ \square]
\end{array}$$

[1] Nowadays, most computers have just plain ole' memory, which you can use to implement your own stack and heap, but these two concepts are so ingrained in the brains of programmers that it's helpful to at least at first maintain the fiction that they are two truly distinct components of a computer.

Notions of Reduction $\rightsquigarrow \subseteq \textsc{Redex} \times \textsc{Term}$:

$$\begin{aligned}
\mathsf{if\ true\ then}\ t_2\ \mathsf{else}\ t_3 &\rightsquigarrow t_2\\
\mathsf{if\ false\ then}\ t_2\ \mathsf{else}\ t_3 &\rightsquigarrow t_3\\
\mathsf{true\ and}\ t_2 &\rightsquigarrow t_2\\
\mathsf{false\ and}\ t_2 &\rightsquigarrow \mathsf{false}\\
\mathsf{true\ or}\ t_2 &\rightsquigarrow \mathsf{true}\\
\mathsf{false\ or}\ t_2 &\rightsquigarrow t_2\\
v\ \mathsf{xor}\ v &\rightsquigarrow \mathsf{false}\\
\mathsf{false\ xor\ true} &\rightsquigarrow \mathsf{true}\\
\mathsf{true\ xor\ false} &\rightsquigarrow \mathsf{true}\\
\mathsf{not\ true} &\rightsquigarrow \mathsf{false}\\
\mathsf{not\ false} &\rightsquigarrow \mathsf{true}
\end{aligned}$$

One-Step Reduction

$$\frac{r \rightsquigarrow t}{E[r] \longrightarrow E[t]}$$

As is common practice, We omit the definition of multi-step reduction. One-step reduction is often omitted too, but it plays a role in our story about abstract machines.

Motivation for Abstract Machine Semantics

The reduction semantics style takes the *structural rules* ofstructural operational semantics, like the rule for if :

$$\frac{t_1 \longrightarrow t_1'}{\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 \longrightarrow \mathsf{if}\ t_1'\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3}$$

and expresses them succinctly as a grammar of *evaluation contexts*.

$$E ::= \ldots \mid E[\mathsf{if}\ \square\ \mathsf{then}\ t\ \mathsf{else}\ t] \ldots$$

We can then view the process of evaluation as a succession of breaking a program down into an evaluation context and redex and applying a notion of reduction:

$$t_0 = E_0[r_0] \longrightarrow E_0[t_0'] = t_1 = E_1[r_1] \longrightarrow E_1[t_1'] = t_2 = E_2[r_2] \longrightarrow \ldots$$

After each reduction step $E_i[r_i] \longrightarrow E_i[t_i'] = t_{i+1}$, the next step begins with asserting the existence of some different from-scratch decomposition of the program $t_{i+1} = E_{i+1}[r_{i+1}]$ to find the next redex to reduce. The sufficiency of this multi-step evaluaiton process typically relies on the truth of a particular theorem.

Theorem 1 (Unique Decomposition)**.** *For any program t, either t is a value v, or there exists a unique context E and unique redex r such that $t = E[r]$.*

Proving this theorem is somewhat involved: when set up appropriately, it's tedious but routine. The existence clause of the theorem guarantees that evaluation is either finished (when t is a value), can make progress ($t = E[r]$ for some context E and redex r that can be reduced according to some notion of reduction (if the semantics is defined for all well-formed programs).

The uniqueness clause of this theorem guarantees that the programming language is deterministic, because if there is an E and r such that $t = E[r]$, then both E and r are unique: there is only one possible step at this point in the program.

Reduction semantics are nice and abstract, but for some purposes, they can be seen as too abstract. Just knowing that unique decomposition holds isn't enough: to evaluate a program, we want not only to know that a decomposition exists: we want to *find* that unique decomposition.

A language implementation modeled after reduction semantics would have to do the work of decomposing and then making a reduction step. Naturally a sufficient proof of unique decomposition contains within it an algorithm to find unique context-redex pairs, but the most straightforward proof technique implies a rather inefficient (but informative) algorithm. In fact, simply solving the decomposition problem is not quite satisfactory. Both reduction semantics and structural operational semantics are woefully inefficient in that each step of program reduction rebuilds the entire program via plugging, and then decomposes it again (which manifests in the reduction example above as $E_0[t_1'] = t_1 = E_1[r_1]$). Most of the time E_0 and E_1 share a great deal of the same structure. A real language implementation need not be so inefficient: it can work directly with E_0 and t_1' to find E_1 and r_1.

Abstract Machine Semantics

We now show how a reduction semantics can be elaborated into a more realistic model of program execution, one that does not keep decomposing and rebuilding a program after each reduction step. An *abstract machine semantics* presents a high-level model of how a computer might actually go about running a program. Historically, abstract machines were the first kind of operational semantics, introduced by Peter Landin in the highly influential 1964 paper *The Mechanical Evaluation of Expressions* [Landin, 1964]. However, abstract machines were considered too low-level by many researchers, and this led to the development of structural operational semantics (by Plotkin) and reduction semantics (by Felleisen). However, it was observed that a reduction semantics is just an abstract representation of an abstract machine.

In this section we expand the reduction semantics above to form an abstract machine. This machine implements the deterministic search for a context-redex pair that can be reduced. Then, after a reduction, the machine proceeds directly from the current decomposition $E_n[t_n']$ to find the next redex, without completely plugging the term and starting over from scratch. This approach is much more efficient in practice.

The abstract machine is defined by a set of *configurations*, which for our simple language are comprised of evaluation context-term pairs. The machine operates in four different *modes*, which are reflected in a grammar of configurations:

$$\begin{array}{lcl} C \in \textsc{Cfg} & & \\ C & ::= & \langle E, t\rangle_{\mathrm{focus}} \mid \langle E, r\rangle_{\mathrm{reduce}} \mid \langle E, v\rangle_{\mathrm{return}} \mid v \end{array}$$

The *focus* mode represents when the abstract machine is searching downward into a term for a redex. The *reduce* mode represents when the abstract machine has found a redex and is ready to reduce it. The *return* mode represents when the machine has found or produced a value, and is returning that value to the current context in order to find the next piece of work to do. The final mode, a standalone value v, denotes that reduction has concluded with a final value.

The heart of the abstract machine semantics is the single step relation

$$\longrightarrow \subseteq \mathrm{CFG} \times \mathrm{CFG}$$

which captures how the machine configurations evolve over time. Since our language happens to be deterministic, we can consider $\longrightarrow$ to be a partial function, but in general that is not the case, so we simply consider it to be a binary relation.

We break the step relation down based on which mode it is in. The *focus* steps determine how to transition by analyzing the top-level structure of the machine's term position.

$$\begin{array}{rcl} \langle E, \mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3\rangle_{\mathrm{focus}} & \longrightarrow & \langle E[\mathsf{if}\ []\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3], t_1\rangle_{\mathrm{focus}} \\ \langle E, t_1\ \mathsf{and}\ t_2\rangle_{\mathrm{focus}} & \longrightarrow & \langle E[[]\ \mathsf{and}\ t_2], t_1\rangle_{\mathrm{focus}} \\ \langle E, t_1\ \mathsf{or}\ t_2\rangle_{\mathrm{focus}} & \longrightarrow & \langle E[[]\ \mathsf{or}\ t_2], t_1\rangle_{\mathrm{focus}} \\ \langle E, t_1\ \mathsf{xor}\ t_2\rangle_{\mathrm{focus}} & \longrightarrow & \langle E[[]\ \mathsf{xor}\ t_2], t_1\rangle_{\mathrm{focus}} \\ \langle E, \mathsf{not}\ t\rangle_{\mathrm{focus}} & \longrightarrow & \langle E[\mathsf{not}\ t], t\rangle_{\mathrm{focus}} \\ \langle E, v\rangle_{\mathrm{focus}} & \longrightarrow & \langle E, v\rangle_{\mathrm{return}} \end{array}$$

In essence, focus steps facilitate the search for a redex by analyzing the structure of a term t step-by-step, leaving a trail of breadcrumbs behind it as extensions to the evaluation context E. Upon discovering a value, it transitions to a *return* configuration that has the same context and term, which we now know to be a value.

The *return* steps determine how to transition by analyzing the top-level structure of the machine's context, which corresponds to the innermost frame, if any.

$$\begin{array}{rcl}
\langle [\,], v\rangle_{\text{return}} & \longrightarrow & v \\
\langle E[\mathsf{if}\ [\,]\ \mathsf{then}\ t\ \mathsf{else}\ t], v\rangle_{\text{return}} & \longrightarrow & \langle E, \mathsf{if}\ v\ \mathsf{then}\ t\ \mathsf{else}\ t\rangle_{\text{reduce}} \\
\langle E[[\,]\ \mathsf{and}\ t], v\rangle_{\text{return}} & \longrightarrow & \langle E, v\ \mathsf{and}\ t\rangle_{\text{reduce}} \\
\langle E[[\,]\ \mathsf{or}\ t], v\rangle_{\text{return}} & \longrightarrow & \langle E, v\ \mathsf{and}\ t\rangle_{\text{reduce}} \\
\langle E[[\,]\ \mathsf{xor}\ t], v\rangle_{\text{return}} & \longrightarrow & \langle E[v\ \mathsf{xor}\ [\,]], t\rangle_{\text{focus}} \\
\langle E[v_1\ \mathsf{xor}\ [\,]], v\rangle_{\text{return}} & \longrightarrow & \langle E, v_1\ \mathsf{xor}\ v_2\rangle_{\text{reduce}} \\
\langle E[\mathsf{not}\ [\,]], v\rangle_{\text{return}} & \longrightarrow & \langle E, \mathsf{not}\ v\rangle_{\text{reduce}}
\end{array}$$

Here is where using inside-out contexts helps us: they are naturally suited for easily examining or updating the innermost *frame* of the evaluation context. If reduction has arrived at a value, and the evaluation context is exhausted, then there is no more work to do: reduction concludes by producing that value. In the case of and , or , and not , a value in the first (or only) position justifies a reduction step, thanks to short-circuit evaluation, so the machine transitions to a *reduce* configuration with the relevant redex in place. In the case of xor , both arguments must be values in order to reduce it, but if the evaluation context is $E[[\,]\ \mathsf{xor}\ t]$, then the machine cannot yet be sure that the second argument is a value. As a result, the machine replaces the innermost frame with a new frame that records the discovered knowledge that the first argument is a value, and proceeds to explore the second argument using a focus configuration. However, if the return configuration's evaluation context is $E[v\ \mathsf{xor}\ [\,]]$, then both arguments to xor are values and reduction can now proceed. The two return steps for xor demonstrate how an evaluation context records information about the current program as the machine traverses it in search of a redex. It's worth observing that the return configuration itself records, albeit temporarily, the discovery of a value v, and preserves that information as it inspects the context.

The *reduce* steps determine how to transition by analyzing the relevant structure of the redex. They proceed by performing the appropriate notion of reduction from the reduction semantics.

$$\begin{array}{rcl}
\langle E, \mathsf{if}\ \mathsf{true}\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2\rangle_{\text{reduce}} & \longrightarrow & \langle E, t_1\rangle_{\text{focus}} \\
\langle E, \mathsf{if}\ \mathsf{false}\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2\rangle_{\text{reduce}} & \longrightarrow & \langle E, t_2\rangle_{\text{focus}} \\
\langle E, \mathsf{true}\ \mathsf{and}\ t_2\rangle_{\text{reduce}} & \longrightarrow & \langle E, t_2\rangle_{\text{focus}} \\
\langle E, \mathsf{false}\ \mathsf{and}\ t_2\rangle_{\text{reduce}} & \longrightarrow & \langle E, \mathsf{false}\rangle_{\text{focus}} \\
\langle E, \mathsf{true}\ \mathsf{or}\ t_2\rangle_{\text{reduce}} & \longrightarrow & \langle E, \mathsf{true}\rangle_{\text{focus}} \\
\langle E, \mathsf{false}\ \mathsf{or}\ t_2\rangle_{\text{reduce}} & \longrightarrow & \langle E, t_2\rangle_{\text{focus}} \\
\langle E, v\ \mathsf{xor}\ v\rangle_{\text{reduce}} & \longrightarrow & \langle E, \mathsf{false}\rangle_{\text{focus}} \\
\langle E, \mathsf{false}\ \mathsf{xor}\ \mathsf{true}\rangle_{\text{reduce}} & \longrightarrow & \langle E, \mathsf{true}\rangle_{\text{focus}} \\
\langle E, \mathsf{true}\ \mathsf{xor}\ \mathsf{false}\rangle_{\text{reduce}} & \longrightarrow & \langle E, \mathsf{true}\rangle_{\text{focus}} \\
\langle E, \mathsf{not}\ \mathsf{false}\rangle_{\text{reduce}} & \longrightarrow & \langle E, \mathsf{true}\rangle_{\text{focus}} \\
\langle E, \mathsf{not}\ \mathsf{true}\rangle_{\text{reduce}} & \longrightarrow & \langle E, \mathsf{false}\rangle_{\text{focus}}
\end{array}$$

Every focus and return machine transition has simply contributed to the search for a redex: plugging the context into the term of a configuration yields the same total program at the beginning and end of any focus and return transition. Not so for reduce, this is where the "real work" gets done.

Each reduce configuration transitions to a focus transition containing the same evaluation context and the outcome of the notion of reduction in term position. The rationale for this design is that the only new information in the outcome transition is the output of reduction, so focusing on that term suffices to find the *next* relevant redex. This design is precisely what avoids the behaviour of reduction semantics that causes it to discard all of the information about the evaluation context and begin decomposition anew.

We can still do a little better, however. Some of our reduction steps tell us enough about reduction such that we can retain even more information. For instance, consider again the transition:

$$\langle E, \mathsf{true}\ \mathsf{or}\ t_2\rangle_{\text{reduce}} \longrightarrow \langle E, \mathsf{true}\rangle_{\text{focus}}$$

Just looking at this transition, we can tell that the next step will be to transition to $\langle E, \mathsf{true}\rangle_{\text{return}}$, because we know that the term position holds a value. As such, we can do better than transitioning to focus by fast-forwarding to the next inevitable step. Olivier Danvy calls these inevitable steps *corridor transitions* because the transitions are forced down a single possible path to some known configuration after some number of steps (here 1). As such, we are wise to update the currently "lossy" transitions to immediately transition to return configurations:

$$\begin{array}{rcl}
\langle E, \textsf{if true then } t_1 \textsf{ else } t_2\rangle_{\text{reduce}} & \longrightarrow & \langle E, t_1\rangle_{\text{focus}} \\
\langle E, \textsf{if false then } t_1 \textsf{ else } t_2\rangle_{\text{reduce}} & \longrightarrow & \langle E, t_2\rangle_{\text{focus}} \\
\langle E, \textsf{true and } t_2\rangle_{\text{reduce}} & \longrightarrow & \langle E, t_2\rangle_{\text{focus}} \\
\langle E, \textsf{false and } t_2\rangle_{\text{reduce}} & \longrightarrow & \langle E, \textsf{false}\rangle_{\text{return}} \\
\langle E, \textsf{true or } t_2\rangle_{\text{reduce}} & \longrightarrow & \langle E, \textsf{true}\rangle_{\text{return}} \\
\langle E, \textsf{false or } t_2\rangle_{\text{reduce}} & \longrightarrow & \langle E, t_2\rangle_{\text{focus}} \\
\langle E, v \textsf{ xor } v\rangle_{\text{reduce}} & \longrightarrow & \langle E, \textsf{false}\rangle_{\text{return}} \\
\langle E, \textsf{false xor true}\rangle_{\text{reduce}} & \longrightarrow & \langle E, \textsf{true}\rangle_{\text{return}} \\
\langle E, \textsf{true xor false}\rangle_{\text{reduce}} & \longrightarrow & \langle E, \textsf{true}\rangle_{\text{return}} \\
\langle E, \textsf{not false}\rangle_{\text{reduce}} & \longrightarrow & \langle E, \textsf{true}\rangle_{\text{return}} \\
\langle E, \textsf{not true}\rangle_{\text{reduce}} & \longrightarrow & \langle E, \textsf{false}\rangle_{\text{return}}
\end{array}$$

In some other languages, corridor transitions will lead the machine to immediately extend the evaluation context, or transition immediately back to reduction. The outcome depends on the properties of the language and its reduction semantics. The natural default, though, is to focus on the outcome of the notion of reduction.

Now, given the definition of $\longrightarrow$, we can specify the corresponding evaluator for this language. The initial state of the machine has an empty context, because we have not yet analyzed the program, and the entire program is in focus position, ($\langle [], t\rangle_{\text{focus}}$). The end state of the machine is represented by a final value v, which must have arisen from the machine is returning a final value to an empty evaluation context (i.e., $\langle [], v\rangle_{\text{return}}$). Given these, the definition of the evaluator follows:

$$\mathit{eval}(t) = v \text{ iff } \langle [], t\rangle_{\text{focus}} \longrightarrow^{*} v.$$

Other programming languages may have additional possible outcomes (e.g. error states), and the evaluator would account for them.

Notice how the evaluation context literally plays the role of the *runtime stack* that you probably learned about in an early programming class. The basic structure of any computation is that when the machine is in state $\langle E, t\rangle_{\text{focus}}$, it is about to analyze t all the way to some value v, which will be returned to the current stack as $\langle E, v\rangle_{\text{return}}$. In big-step semantics, we can see the part of the same phenomenon by following the big-step reduction tree upwards. An arithmetic summation rule of the form

$$\frac{t_1 \Downarrow n_1 \qquad t_2 \Downarrow n_2}{t_1 + +t_2 \Downarrow n_3} \text{ where } n_3 = n_1 + n_2$$

first evaluates t_1 all the way down to n_1, "remembering" that the next thing to do is evaluate t_2, before summing the results. This is made much more explicit in the abstract machine semantics, since the evaluation context is explicitly carrying this memory. And this is why abstract machine semantics are a clear way of expressing programming language control-constructs. Language features that implement nontraditional control patterns can be expressed by manipulating the evaluation context. We'll see examples of this in the near future.

Bibliography

P. Aczel. An introduction to inductive definitions. In J. Barwise, editor, *Handbook of Mathematical Logic*, volume 90 of *Studies in Logic and the Foundations of Mathematics*, chapter C.7, pages 739–782. North-Holland, 1977.

J. Avigad. Reliability of mathematical inference. *Synthese*, 2020. doi: 10.1007/s11229-019-02524-y. `https://doi.org/10.1007/s11229-019-02524-y`.

F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, New York, NY, USA, 1998. ISBN 0-521-45520-0.

J. Bagaria. Set theory. In E. N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. The Metaphysics Research Lab, winter 2014 edition, 2014. URL `http://plato.stanford.edu/archives/win2014/entries/set-theory/`.

R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi. A survey of symbolic execution techniques. *ACM Comput. Surv.*, 51(3):50:1–50:39, May 2018. ISSN 0360-0300. doi: 10.1145/3182657. URL `http://doi.acm.org/10.1145/3182657`.

H. P. Barendregt. *The lambda calculus - its syntax and semantics*, volume 103 of *Studies in logic and the foundations of mathematics*. North-Holland, 1985. ISBN 978-0-444-86748-3.

A. Bauer. Proof of negation and proof by contradiction. Mathematics and Computation Blog, March 2010. `http://math.andrej.com/2010/03/29/proof-of-negation-and-proof-by-contradiction/`.

L. Crosilla. Set Theory: Constructive and Intuitionistic ZF. In E. N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, Summer 2020 edition, 2020.

M. Felleisen and D. P. Friedman. A syntactic theory of sequential state. *Theoretical Computer Science*, 69(3):243–287, 1989. ISSN 0304-3975. doi: https://doi.org/10.1016/0304-3975(89)90069-8. URL `http://www.sciencedirect.com/science/article/pii/0304397589900698`.

M. Felleisen, R. B. Findler, and M. Flatt. *Semantics Engineering with PLT Redex*. MIT Press, 1st edition, 2009.

J. Ferreirós. The early development of set theory. In E. N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, summer 2019 edition, 2019. URL `https://plato.stanford.edu/archives/sum2019/entries/settheory-early/`.

D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in cyclone. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 282–293, New York, NY, USA, 2002. ACM. ISBN 1-58113-463-0. doi: 10.1145/512529.512563. URL `http://doi.acm.org/10.1145/512529.512563`.

P. R. Halmos. *Naive Set Theory*. Springer-Verlag, first edition, Jan. 1960. ISBN 0387900926.
A classic introductory textbook on set theory.

P. R. Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, New York, NY, USA, 2012. URL `http://www.cs.cmu.edu/%7Erwh/plbook/1sted-revised.pdf`.

D. J. Howe. Proving congruence of bisimulation in functional programming languages. *Inf. Comput.*, 124 (2):103–112, Feb. 1996. ISSN 0890-5401.

G. Kahn. Natural semantics. In *4th Annual Symposium on Theoretical Aspects of Computer Sciences on STACS 87*, pages 22–39, London, UK, UK, 1987. Springer-Verlag.

R. Krebbers. *The C standard formalized in Coq.* PhD thesis, Radboud University Nijmegen, December 2015. URL https://robbertkrebbers.nl/thesis.html.

C. Kuratowski. Sur la notion de l'ordre dans la théorie des ensembles. *Fundamenta Mathematicae*, 2(1): 161–171, 1921. doi: 10.4064/fm-2-1-161-171. https://web.archive.org/web/20190429103938/http://matwbn.icm.edu.pl/ksiazki/fm/fm2/fm2122.pdf.

P. J. Landin. The mechanical evaluation of expressions. *Comput. J.*, 6(4):308–320, 1964. doi: 10.1093/comjnl/6.4.308. URL https://doi.org/10.1093/comjnl/6.4.308.

X. Leroy and H. Grall. Coinductive big-step operational semantics. *Inf. Comput.*, 207(2):284–304, Feb. 2009. ISSN 0890-5401.

P. Maddy. Believing the axioms. I. *The Journal of Symbolic Logic*, 53(02):481–511, 1988.
An interesting (though complicated) analysis of why set theorists believe in their axioms.

J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part I. *Commun. ACM*, 3(4):184–195, 1960.

J. H. J. Morris. *Lambda-Calculus Models of Programming Languages.* PhD thesis, Massachusetts Institute of Technology, Feb. 1969. URL http://hdl.handle.net/1721.1/64850.

A. M. Pitts. *Nominal sets: Names and symmetry in computer science.* Cambridge University Press, 2013.

G. D. Plotkin. The origins of structural operational semantics. *J. Log. Algebr. Program.*, 60-61:3–15, 2004a. doi: 10.1016/j.jlap.2004.03.009. URL http://dx.doi.org/10.1016/j.jlap.2004.03.009.

G. D. Plotkin. A structural approach to operational semantics. *J. Log. Algebr. Program.*, 60-61:17–139, 2004b.

B. Popik. "pull yourself up by your bootstraps". Weblog entry, September 2012. https://www.barrypopik.com/index.php/new_york_city/entry/pull_yourself_up_by_your_bootstraps/.

E. Reck and G. Schiemer. Structuralism in the Philosophy of Mathematics. In E. N. Zalta, editor, *The Stanford Encyclopedia of Philosophy.* Metaphysics Research Lab, Stanford University, Spring 2020 edition, 2020.

D. Sangiorgi. On the origins of bisimulation and coinduction. *ACM Trans. Program. Lang. Syst.*, 31(4): 15:1–15:41, May 2009. ISSN 0164-0925.

W. Sieg and D. Schlimm. Dedekind's analysis of number: Systems and axioms. *Synthese*, 147(1):121–170, Oct 2005.

J. Spolsky. The law of leaky abstractions. Joel on Software Blog, November 2002. https://www.joelonsoftware.com/2002/11/11/the-law-of-leaky-abstractions/.

G. L. Steele. Debunking the "expensive procedure call"" myth or, procedure call implementations considered harmful or, lamdba: The ultimate goto. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1977. URL http://dspace.mit.edu/handle/1721.1/5753.

S. Stenlund. Descriptions in intuitionistic logic. In S. Kanger, editor, *Proceedings of the Third Scandinavian Logic Symposium*, volume 82 of *Studies in Logic and the Foundations of Mathematics*, pages 197 – 212. Elsevier, 1975. URL http://www.sciencedirect.com/science/article/pii/S0049237X08707328.

M. Tiles. Book Review: Stephen Pollard. Philosophical Introduction to Set Theory. *Notre Dame Journal of Formal Logic*, 32(1):161–166, 1990.
A brief introduction to the philosophical issues underlying set theory as a foundation for mathematics.

C. Urban and M. Norrish. A formal treatment of the Barendregt variable convention in rule inductions. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Mechanized Reasoning about Languages with Variable Binding*, MERLIN '05, page 25–32, New York, NY, USA, 2005. Association for Computing Machinery. ISBN 1595930728. doi: 10.1145/1088454.1088458. URL `https://doi.org/10.1145/1088454.1088458`.

C. Urban, S. Berghofer, and M. Norrish. Barendregt's variable convention in rule inductions. In *Proceedings of the 21st International Conference on Automated Deduction: Automated Deduction*, CADE-21, page 35–50, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 9783540735946. doi: 10.1007/978-3-540-73595-3_4. URL `https://doi.org/10.1007/978-3-540-73595-3_4`.

D. van Dalen. *Logic and structure (3. ed.)*. Universitext. Springer, 1994. ISBN 978-3-540-57839-0.

A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94, Nov. 1994.

B. Zimmer. figurative "bootstraps". email to linguistlist mailing list, August 2005. `http://listserv.linguistlist.org/pipermail/ads-l/2005-August/052756.html`.