Chapter 7

Big-Step Semantics

By now, you have learned how to:

1. define sets using *inductive definitions*;
2. prove universal properties of inductively-defined sets using the corresponding principle of derivation induction (and rule induction) (a.k.a. *proof by induction*); and
3. define total functions that map an inductively defined set to other sets using the corresponding principle of recursive definition.

Isn't induction great?!? Given the last of these tools, we can define total functions over recursively defined sets with relative ease. In fact, we can use this principle to define an evaluator for the language of Boolean Expressions (see below).

As our languages get more sophisticated, though, we will find that definition by recursion does not always suffice for defining and analyzing semantics. Some programming language definitions have properties that don't play well with total functions:

1. If your language has programs that *don't terminate*, then you will be hard-pressed to define them recursively;
2. If your language has *nondeterministic behaviour*, meaning that one program or program fragment can produce more than one possible behaviour, then a recursive function definition may awkwardly describe its semantics: a more general relation between programs and possible results may be a more natural specification;
3. Sometimes you want to define a *partial function* somehow related to your language, and an equational definition of this can also be awkward some times.

This set of notes introduces a common approach to defining semantics that addresses some of these issues. We will do so while simultaneously extending our language of Boolean expressions with *arithmetic expressions*. This is not strictly necessary, but introduces some new language concepts (especially *run-time errors*) along the way. We will use a different technique than recursion to define the evaluator, but the technique we are introducing is actually hidden inside the proof of the principle of recursive definition. Let's tease that out.

7.1 Boolean Expression Function Revisited

Earlier, we learned the Principle of Definition by Recursion, which enables us to define total functions over inductively defined sets, like the terms of our language of Boolean Expressions. One particular function over

Boolean expressions we defined was its evaluator.

$$\begin{gathered}
\textsc{Pgm} = \textsc{Term}, \quad \textsc{Obs} = \textsc{Value} \\
eval : \textsc{Pgm} \to \textsc{Obs} \\
\begin{aligned}
eval(\mathsf{true}) &= \mathsf{true} \\
eval(\mathsf{false}) &= \mathsf{false} \\
eval(\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3) &= eval(t_2) \text{ if } eval(t_1) = \mathsf{true} \\
eval(\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3) &= eval(t_3) \text{ if } eval(t_1) = \mathsf{false}
\end{aligned}
\end{gathered}$$

Recall that this evaluator definition is justified by the principle of recursive definition from $t \in \textsc{Term}$, which means that we chose:

1. $S = \textsc{Value}$;
2. $s_t = \mathsf{true}$;
3. $s_f = \mathsf{false}$;
4. $H_{if} : \textsc{Value} \times \textsc{Value} \times \textsc{Value} \to \textsc{Value}$
   $H_{if}(\mathsf{true}, v_1, v_2) = v_1$
   $H_{if}(\mathsf{false}, v_1, v_2) = v_2$.

and fed them into the principle to produce a unique function, and we then convince ourselves intuitively that this is in fact our intended evaluator.

If we consider the *proof* of the Principle of Definition by Recursion, however, we see that the first step of the proof is to construct a binary relation $\Downarrow \subseteq \textsc{Term} \times \textsc{Value}$, typically called a *big-step* relation, that we then externally prove is a total function, satisfies the equations given, and is unique.[1] In short, we explicitly constructed our function as a binary relation, and then proved that it's a function. Let's examine that explicit construction.

If we inline our chosen elements into the rules for $\Downarrow$, we get roughly the following:

$$\frac{}{\mathsf{true} \Downarrow \mathsf{true}}\ (\text{etrue}) \qquad \frac{}{\mathsf{false} \Downarrow \mathsf{false}}\ (\text{efalse})$$

$$\frac{t_1 \Downarrow \mathsf{true} \quad t_2 \Downarrow v_2}{\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 \Downarrow v_2}\ (\text{eif-t}) \qquad \frac{t_1 \Downarrow \mathsf{false} \quad t_3 \Downarrow v_3}{\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 \Downarrow v_3}\ (\text{eif-f})$$

The (eif-t) and (eif-f) rules together specialize the following single rule with respect to the definition of H_{if}:

$$\frac{t_1 \Downarrow v_1 \quad t_2 \Downarrow v_2 \quad t_3 \Downarrow v_3}{\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 \Downarrow H_{if}(v_1, v_2, v_3)}\ (\text{eif})$$

In short, it's not hard to prove that if

$$\begin{gathered}
\mathcal{R}_{\Downarrow} = \bigcup \{\, \text{etrue,efalse,eif-t,eif-f} \,\}, \\
\mathcal{R}' = \bigcup \{\, \text{etrue,efalse,eif} \,\}
\end{gathered}$$

then

$$\begin{aligned}
\Downarrow &= \{\, \langle t, v \rangle \in \textsc{Term} \times \textsc{Value} \mid \exists \mathcal{D} \in \textsc{Deriv}[\mathcal{R}_{\Downarrow}].\mathcal{D} :: \langle t, v \rangle \,\} \\
&= \{\, \langle t, v \rangle \in \textsc{Term} \times \textsc{Value} \mid \exists \mathcal{D} \in \textsc{Deriv}[\mathcal{R}'].\mathcal{D} :: \langle t, v \rangle \,\}.
\end{aligned}$$

The key observation is that based on the value of v_1, H_{if} discards either v_2 or v_3, so there is no real need for the discarded derivation. This improvement in the rules is just due to some human cleverness: it is not

[1]We use the name $\Downarrow$ for reasons that should be explained shortly.

fundamental, but it also matches the last two equations of our recursive definition above. Later, we use this split to motivate a nice implementation of the language.

Based on our proof of the Principle of Definition by Recursion, it is clear that $\Downarrow = eval$, both count as definitions of the same total function. However it's not unusual to write out *eval* equationally. Some other languages that we define using big-step will need an equational definition to handle different possible outcomes (e.g. non-termination).

$$\begin{array}{l} \text{PGM} = \text{TERM}, \quad \text{OBS} = \text{VALUE} \\ eval : \text{PGM} \to \text{OBS} \\ eval(t) = v \text{ if } t \Downarrow v \end{array}$$

The key difference between the two approaches is a matter of flexibility. Inductive definitions are more general: they need not be total, and they need not be (partial) functions (i.e., deterministic). We sometimes want that flexibility in the definition of our language semantics. For this reason, among others, inductive definitions like $\Downarrow$ are often the preferred mode of specifying the semantics of programming language.

7.2 Backward Reasoning

One of the nice things about a recursive function definition is that we can use the equations that define the function to calculate what a recursive function maps its argument to. However, if we didn't define the B language as a set of recursive equations, then we would need a new strategy for calculating the results of evaluation (either by hand, or by implementing an interpreter).

Before looking at the general strategy, lets consider an instance. How do we determine what (if anything) if false then false else true evaluates to? Given the structure of our evaluators above, we know that we must determine what it big-steps to. Since the $\Downarrow$ relation is defined inductively, we know that if false then false else true $\Downarrow v$ for some v if and only if there is some derivation

$$\mathcal{D} :: \textsf{if false then false else true} \Downarrow v$$

. Thus, calculating the mapping boils down to *searching* for a derivation that begins with our term:

$$\begin{array}{c} \vdots \\ \textsf{if false then false else true} \Downarrow ??? \end{array}$$

Technically we must consider all of the rules (this is backward reasoning). We quickly (so quickly that we may not even notice that we did) rule out (pun intended) (etrue), because if false then false else true $\neq$ true and (efalse), for the same reason. This leaves us with two remaining possibilities, (eif-t) and (eif-f). Which one to try? Who knows! Let's be systematic and try them in order, starting with (eif-t):

$$\dfrac{\begin{array}{c}\vdots\\ \textsf{false} \Downarrow \textsf{true}\end{array} \quad \begin{array}{c}\vdots\\ \textsf{false} \Downarrow v_2\end{array}}{\textsf{if false then false else true} \Downarrow v_2} \text{ (eif-t)},$$

We're just following our noses, filling in those parts of the (eif-t) rule that are determined by the structure of the rule and the information that we already have in the conclusion. We don't know what the final value would be, but based on the rule, it will be some value, in particular the result of big-stepping true, the consequent position of the if expression.

We've got two options for searching, the two premises. Let's work left to right. Doh! We're stuck!

$$\dfrac{\begin{array}{c}X\\ \textsf{false} \Downarrow \textsf{true}\end{array} \quad \begin{array}{c}\vdots\\ \textsf{false} \Downarrow v_2\end{array}}{\textsf{if false then false else true} \Downarrow v_2} \text{ (eif-t)},$$

There is no rule whose conclusion matches false $\Downarrow$ true, which arose when we chose (eif-t). We *could* try to evaluate the other premise, but that would be a waste of energy because we simply cannot complete this derivation. So we have to try something else. Our only remaining option if (eif-f), so if *that* doesn't work, then we know that the term does not big-step.

$$\dfrac{\begin{array}{cc}\vdots & \vdots\\ \mathsf{false} \Downarrow \mathsf{false} & \mathsf{true} \Downarrow v_3\end{array}}{\mathsf{if\ false\ then\ false\ else\ true} \Downarrow v_3}\,(\text{eif-f}),$$

This time we *can* make progress on the left premise, finishing it off with an application of (efalse). Along the way we might have briefly considered and rejected (etrue), (eif-t), and (eif-f).

$$\dfrac{\dfrac{}{\mathsf{false} \Downarrow \mathsf{false}}\,(\text{efalse}) \qquad \begin{array}{c}\vdots\\ \mathsf{true} \Downarrow v_3\end{array}}{\mathsf{if\ false\ then\ false\ else\ true} \Downarrow v_3}\,(\text{eif-f}),$$

And with a little more consideration we find that (etrue) finishes off this derivation.

$$\dfrac{\dfrac{}{\mathsf{false} \Downarrow \mathsf{false}}\,(\text{efalse}) \qquad \dfrac{}{\mathsf{true} \Downarrow \mathsf{true}}\,(\text{etrue})}{\mathsf{if\ false\ then\ false\ else\ true} \Downarrow \mathsf{true}}\,(\text{eif-f}),$$

Now we know for sure that $\mathsf{if\ false\ then\ false\ else\ true} \Downarrow \mathsf{true}$: our derivation is a "proof" of this. Can it evaluate to any other values? We can show (by exhaustion) that this is the only derivation tree that we can build. Naturally we can prove that this is the case for every TERM t.

So our equational definition of this language allowed us to reason about values using equational reasoning. Here we reason by *bottom-up proof search*. We did so informally, but we can make this reasoning process more concrete. Given a term t, we try to find a value that it big-steps to by searching bottom-up for a derivation that has t on its left-hand side, considering the rules that could possibly be instantiated to match our goal. This reasoning is made formal as a set of backward-reasoning propositions. To make them a little more specialized, we write lemmas that *distinguish* the top-level structure of the term under consideration. Such principles are quite useful, and we intuitively make these leaps of reasoning, though they can be stated and proven explicitly.

Proposition 33 (Backward Reasoning, distinguishing t)**.**

1. *If* $\mathsf{true} \Downarrow v$ *then* $v = \mathsf{true}$*.*

2. *If* $\mathsf{false} \Downarrow v$ *then* $v = \mathsf{false}$*.*

3. *If* $\mathsf{if}\, t_1\ \mathsf{then}\, t_2\ \mathsf{else}\, t_3 \Downarrow v$ *then either*

   (a) $t_1 \Downarrow \mathsf{true}$ *and* $t_2 \Downarrow v$ *or*

   (b) $t_1 \Downarrow \mathsf{false}$ *and* $t_3 \Downarrow v$ *.*

To prove these propositions, we first expand them to be formal statements about derivations. For example, item 3 expands to the following:

Proposition 34. $\forall \mathcal{D}.\mathcal{D} :: \mathsf{if}\, t_1\ \mathsf{then}\, t_2\ \mathsf{else}\, t_3 \Downarrow v \Rightarrow (\exists \mathcal{E}_1, \mathcal{E}_2.\mathcal{E} :: t_1 \Downarrow \mathsf{true} \wedge \mathcal{E}_2 :: t_2 \Downarrow v) \vee (\exists \mathcal{E}_1, \mathcal{E}_3.\mathcal{E} :: t_1 \Downarrow \mathsf{false} \wedge \mathcal{E}_3 :: t_3 \Downarrow v)$.

Proof. By cases on the last rule used to form $\mathcal{D}$.

Case 20 (etrue)*.* Vacuous because true does not match if .

Case 21 (efalse)*.* Vacuous.

Case 22 (eif-t)*. Exercise!*

Case 23 (eif-f)*. Exercise!*

□

Each backward reasoning principle can be *devised* by staring at the rules. Typical phrasings have one case for each term construct, and then the proposition incorporates, using disjunction, each of the possible rules. If no rule applies, then the conclusion is $\perp$: the zero-ary case of disjunction.

Each backward reasoning principle can be proven in the same manner as this one: by cases on the derivation $\mathcal{D}$. The proof is straightforward, especially since most of the cases are vacuous because the rule doesn't match. The proof is simple enough that many books or papers do not even bother to state the proof strategy or its cases. However, it is important that you know how one *would* prove such a thing. While it's intuitively obvious to us, it would not be intuitively to a computer. Nonetheless we can make the proof a purely mechanical thing that a mechanized proof assistant on a computer could check for correctness.

As you will see, these propositions can serve as the basis for an implementation of the language.

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`.