lec2: Rule-based semantics; induction

Jana Dunfield

January 12, 2022

1 "Programming Languages"

What is a programming language? Agreement on a precise definition is elusive, but we can define a programming language as: a well-defined way to instruct computers, using symbols.

If computers compute (do computations), then a programming language is a precise, symbolic description of a set of possible computations.

Caveats:

- "Symbolic": there have been occasional attempts at visual PLs (Smalltalk-80 and Logo are not really visual, but languages like Prograph, developed in the 1980s and 1990s, were certainly intended to be visual), mirroring occasional attempts at diagrammatic logics.
- "Precise": the vast majority of programming languages have never been described precisely.

The second caveat is unfortunate:

- Programmers need precision so they know how to reason about their programs.
- Language implementors need precision so they know how to implement (interpret, compile, translate to another language) a language.
- Unfortunately, most PLs are defined using English; a few are defined using math/logic. See "The C language does not exist" (Bessey et al., 2010).

How do we precisely define a programming language?

- **Syntax** describes *which sequences of symbols are reasonable*.
- **Dynamic semantics** describes *how to run programs*.
- **Static semantics** describes *what programs are*.

2 Rule-based semantics

Rules will define the dynamic semantics for a very tiny programming language. This language can do only one thing: add integers.

The rule notation was invented in the 1930s by Gerhard Gentzen (more on him later, unfortunately), for formal logic. This notation was used in CISC 204; the logical rules presented there are very similar to those Gentzen developed. Programming languages researchers use Gentzen's notation extensively for purposes other than logic. (We will cover formal logic in 465/865, though in a somewhat different way from 204.) You should not need to know or remember the rules from 204 to follow these notes.

A rule looks like

$$\frac{\textit{premise 1} \qquad \cdots \qquad \textit{premise n}}{\textit{conclusion}}\ \text{name of rule}$$

where *premise 1* through *premise n* are the premises of the rule (n can be zero, as we'll see shortly) and *conclusion* is its conclusion. The rule means:

If every premise is true, then the conclusion is true.

What counts as "true" is defined by whatever system of rules we are using at the moment. (The kind of logic learned 204 is just one kind of logic, and logics are just one kind of formal system that we can define with rules.) So it's often better to think of "true" as "derivable":

If every premise is derivable using our rules,
then the conclusion is derivable using our rules.

Which rules count as "our rules" changes over time and can be a source of confusion, though hopefully it will be less confusing this term, since I'm unable to rapidly edit and erase rules on the board during lecture.

We will define what a given expression, say (+ 2 2), *steps to*. For example, (+ 2 2) should step to 4, because we want + to mean "add these two expressions".

The slightly more complicated expression (+ 2 (+ 3 4)) should eventually "give" the result 9, since $9 = 2 + (3 + 4)$, but the idea of stepping is to do one operation at a time. So (+ 2 (+ 3 4)) does not step to 9, but to (+ 2 7), where (+ 3 4) is replaced by 7.

That is, we want

$$\texttt{(+ 2 (+ 3 4))} \mapsto \texttt{(+ 2 7)}$$

to "be true", or equivalently, to be derivable.

This sequence of symbols, (+ 2 (+ 3 4)) $\mapsto$ (+ 2 7), is called a *judgment*: it "judges" that the left-hand expression steps to the right-hand expression.

To be clear about what counts as an expression, here is a grammar defining expressions:

$$\begin{array}{lrl} \text{expressions} & e ::= & n \\ & \mid & \texttt{(+ } e\ e\texttt{)} \end{array}$$

As mentioned in lec1, the repetition of e in the last production does not mean the two occurrences of e have to be the same.

But this is not really clear yet, because I haven't defined what an n is. Rather than write a grammar, I am going to assume that we know what an integer is, and then say that n is an integer:

$$\begin{array}{lrcl} \text{integers} & n & \in & \mathbb{Z} \\ \text{expressions} & e & ::= & n \\ & & & \mid \texttt{(+ } e\ e\texttt{)} \end{array}$$

(Mathematicians use $\mathbb{Z}$ to mean "the integers" because the German word *Zahlen* starts with a Z.)

Below are the three rules we will be using, until I note otherwise.

The box containing $e \mapsto e'$ indicates that we are about to give all the rules for the judgment "e steps to e'". The phrase "expression e steps to e'" describes in English what we are defining, and tells us how to verbalize (and think about) the judgment. Try not to pronounce the symbol $\mapsto$ as "arrow" or "right arrow": it's better to remind yourself that it's defining "steps to". (Even if you're not quite sure what stepping is yet, "stepping" provides more intuition than "right arrow".)

$\boxed{e \mapsto e'}$ expression e steps to e'

$$\frac{n = n_1 + n_2}{\texttt{(+ } n_1\ n_2\texttt{)} \mapsto n}\ \text{step-add}$$

$$\frac{e_1 \mapsto e_1'}{\texttt{(+ } e_1\ e_2\texttt{)} \mapsto \texttt{(+ } e_1'\ e_2\texttt{)}}\ \text{step-add-left} \qquad \frac{e_2 \mapsto e_2'}{\texttt{(+ } e_1\ e_2\texttt{)} \mapsto \texttt{(+ } e_1\ e_2'\texttt{)}}\ \text{step-add-right}$$

Again: The things above the line are *premises*, and the things below the line are *conclusions*.

Let's go through each of these.

The first rule, step-add, says

For all integers n, n_1, n_2:
If $n = n_1 + n_2$,
then the expression $\texttt{(+ } n_1\ n_2\texttt{)}$ steps to n.

Equivalently, it says:

For all n, n_1, n_2:
If $n = n_1 + n_2$,
then the expression $\texttt{(+ } n_1\ n_2\texttt{)}$ steps to n.

This is the same as the earlier statement but without the word "integers". By defining n in our grammar, we are agreeing that whenever we write n (or n', n'', n_1, n_2, n_3, n_0, etc.), we mean an integer. So writing "integers" is redundant (but I may do it sometimes for clarity).

Also equivalently, it says:

If $n = n_1 + n_2$,
then the expression $\texttt{(+ } n_1\ n_2\texttt{)}$ steps to n.

The n, n_1 and n_2 are called *meta-variables*: they are variables not in the sense of program variables or mathematical variables (like x and y in the equation $x = 1 + y$), but variables at the "meta level" ("meta" means "above", and we are working "above" the level of the expressions we are talking about). In the language of predicate logic (as in CISC 204), meta-variables are always universally quantified.

So the first version of the statement is longer than it needs to be; we don't need to say "for all", because that's how meta-variables always work, and we don't need to say that n, n_1 and n_2 are integers, because we're using a grammar that says n (and any subscripted or superscripted version of n) is an integer.

> For all integers n, n_1, n_2:
> If $n = n_1 + n_2$,
> then the expression $(+\ n_1\ n_2)$ steps to n.

Let's see what we can do with just this rule.

We can step (+ 3 4) to 7. That is, we can derive the judgment

$$(+\ 3\ 4) \mapsto 7$$

We show this by writing a *derivation*, which looks just like the rule except that instead of meta-variables n, n_1, n_2, we have actual integers:

$$\dfrac{7 = 3 + 4}{(+\ 3\ 4) \ \mapsto\ 7}\ \text{step-add}$$

The meaning of the rule is "For all n, n_1, n_2: . . . ". I chose n to be 7, n_1 to be 3, and n_2 to be 4.

$$\dfrac{7 = 3 + 4}{(+\ 3\ 4) \ \mapsto\ 7}\ \text{step-add}$$

Exercise 1. Write a derivation of $(+\ 2\ 1) \mapsto 3$.

What about the premise $7 = 3 + 4$? I am assuming we know how to add integers.

Note that step-add *only* works for expressions $(+\ e_1\ e_2)$ in which both subexpressions e_1 and e_2 are integers. If step-add were our only rule, we would be unable to step larger expressions: we wouldn't be able to step (+ 2 (+ 3 4)) because the second subexpression, (+ 3 4), is not an integer.

(It will *step to* an integer, but that is not the same as *being* an integer.)

The next rule, step-add-left, allows us to step larger expressions in which the first subexpression is not an integer.

$$\dfrac{e_1 \ \mapsto\ e_1'}{(+\ e_1\ e_2) \ \mapsto\ (+\ e_1'\ e_2)}\ \text{step-add-left}$$

However, that doesn't help us step the example expression (+ 2 (+ 3 4)), because 2 does not step to anything. (I didn't give you a rule that says 2 steps to 2, so it doesn't.)

Our last rule, step-add-right, will let us step the example.

$$\dfrac{e_2 \ \mapsto\ e_2'}{(+\ e_1\ e_2) \ \mapsto\ (+\ e_1\ e_2')}\ \text{step-add-right}$$

Here is a derivation of $(+\ 2\ (+\ 3\ 4)) \mapsto (+\ 2\ 7)$:

$$\dfrac{\dfrac{7 = 3 + 4}{(+\ 3\ 4) \ \mapsto\ 7}\ \text{step-add}}{(+\ 2\ (+\ 3\ 4)) \ \mapsto\ (+\ 2\ 7)}\ \text{step-add-right}$$

Conclusion of a derivation The bottom part of a rule is its conclusion. The bottom part of a derivation is also called its conclusion, or sometimes its *concluding judgment*. For example,

$$\dfrac{\dfrac{3 = 1 + 2}{\texttt{(+ 1 2)} \mapsto 3}\ \text{step-add}}{\texttt{(+ (+ 1 2) 11)} \mapsto \texttt{(+ 3 11)}}\ \text{step-add-left}$$

is a derivation that concludes $\texttt{(+ (+ 1 2) 11)} \mapsto \texttt{(+ 3 11)}$.

We would also say that the *concluding rule* of this derivation is step-add-left.

Rule equivalence We could write step-add a little differently (which, to distinguish it, I'll call "add-const"):

$$\dfrac{}{\texttt{(+}\ n_1\ n_2\texttt{)} \ \mapsto\ n_1 + n_2}\ \text{add-const}$$

However, both versions of the rule are *equivalent*, in the sense that *exactly the same judgments are derivable*. The rule step-add has an extra meta-variable n, but the premise $n = n_1 + n_2$ allows only one choice of n for any n_1 and n_2. This is a relatively easy case of equivalence.

Exercise 2. Find e' such that $\texttt{(+ 2 7)} \mapsto e'$.

That is, write a derivation whose conclusion is $\texttt{(+ 2 7)} \mapsto e'$, but where e' is an actual expression, not a meta-variable.

Hint: use the rule step-add.

Exercise 3. In the previous exercise, why couldn't you use step-add-left or step-add-right?

This notation for rules and derivations comes from **?**. CISC 204 uses a horizontal line for rules, but didn't "stack" rule applications to construct derivations.

3 Metatheory

“Theory” is what we defined (using grammars and rules). Metatheory is theory *about our theory*: if we prove something about our operational semantics (or a denotational semantics, type system, etc.), that’s metatheory. Here is our very first meta-theoretic result:

Theorem 1.
For all integers n, it is the case that $(+\ n\ n) \mapsto 2n$.

Proof. Assume n is an integer.
By rule step-add, $(+\ n\ n) \mapsto n + n$.
By a property of arithmetic,

$$n + n = 2n$$

By the above equation,

$$(+\ n\ n) \mapsto 2n \qquad \square$$

I consider the line “Assume n…” to be redundant: we are using a convention that n always stands for an integer, and whenever you want to prove a statement “For all $\cdots$,” you assume that the $\cdots$ are given. (CISC 204 note: This corresponds to using the $\forall$i rule.) Some people (including some of my research collaborators) like to write it out anyway.

Since our three rules step-add, step-add-left, step-add-right define when it is the case that an expression steps to something, the phrase

it is the case that $(+\ n\ n) \mapsto 2n$

is interchangeable with all of the following:

the judgment $(+\ n\ n) \mapsto 2n$ *holds*
the judgment $(+\ n\ n) \mapsto 2n$ *is derivable*
there exists a derivation of $(+\ n\ n) \mapsto 2n$
there exists a derivation $\mathcal{D}$ *of* $(+\ n\ n) \mapsto 2n$
there exists $\mathcal{D}$ *deriving* $(+\ n\ n) \mapsto 2n$
$(+\ n\ n) \mapsto 2n$

I prefer not to write

For all integers n, $(+\ n\ n) \mapsto 2n$.

because it could be hard to tell whether the comma is an ordinary “English” comma, or part of some judgment containing a comma. We haven’t seen such judgments yet, but we will use them later.

Using Theorem 1, we know that $(+\ 3\ 3) \mapsto 6$. Do we know whether $(+\ 3\ 3) \mapsto 7$?

It shouldn’t ($3 + 3$ does not equal 7), and it doesn’t, but Theorem 1 does *not* show that. It says that $(+\ 3\ 3)$ *can* step to 6.

Let’s show that whenever we have an expression $(+\ n\ n)$, it steps to $2n$.

Theorem 2.
For all expressions e', if $\mathcal{D}_1$ derives $(+\ n\ n) \mapsto e'$
then $e' = 2n$.

Proof. The only rule whose conclusion can have the form $(+\ n\ n) \mapsto e'$ is step-add.

Therefore, the concluding rule of $\mathcal{D}_1$ must be step-add, and $e' = n+n$, which is equal to $2n$. □

The property that a given expression e always steps to a particular e' is called *determinism* or *determinacy*.

Conjecture 1.
For all e, e_1, e_2 such that $e \mapsto e_1$ and $e \mapsto e_2$ then $e_1 = e_2$.

When I say $e_1 = e_2$, I mean that e_1 and e_2 are exactly the same expression.

Counterexample: Let $e = (+\ (+\ 1\ 2)\ (+\ 4\ 5))$. Using rule step-add-left, we can derive

$$e \mapsto \underbrace{(+\ 3\ (+\ 4\ 5))}_{e_1}$$

Using rule step-add-right, we can derive

$$e \mapsto \underbrace{(+\ (+\ 1\ 2)\ 9)}_{e_2}$$

This gives us different expressions e_1 and e_2.

■ **Exercise 4.** Write the full derivations of the judgments $e \mapsto e_1$ and $e \mapsto e_2$.

However, if we keep stepping e_1 and e_2 we will get 12 in each case. That suggests a different conjecture based on stepping e *zero or more times*. First, we define "stepping zero or more times" using rules:

$\boxed{e \mapsto^* e'}$ expression e steps zero or more times to e'

$$\frac{}{e \mapsto^* e}\ \text{steps-zero} \qquad\qquad \frac{e \mapsto e_1 \qquad e_1 \mapsto^* e_2}{e \mapsto^* e_2}\ \text{steps-multi}$$

■ **Exercise 5.** Define a judgment $e \mapsto^+ e'$, read "stepping *one* or more times", using rules.

Conjecture 2.
If $e \mapsto^* e_1$ and $e \mapsto^* e_2$
then $e_1 = e_2$.

Unfortunately, this conjecture is also false:

$$(+\ 1\ 2) \mapsto^* (+\ 1\ 2)$$
$$(+\ 1\ 2) \mapsto^* 3$$

We want to talk about the result of stepping "as far as possible", not what happens when we *can* step the expression but choose not to.

For this, we need a notion of *values*—expressions that don't do anything. For the moment, we have a very small language: every expression is either an integer or an addition. The additions should not be values, because they can keep stepping. The integers are values, because they don't (and shouldn't) step.

$$\text{values} \quad v ::= n$$

Now we can state a conjecture that should, in fact, be true:

Conjecture 3.
If $e \mapsto^* v_1$ and $e \mapsto^* v_2$
then $v_1 = v_2$.

(Equivalently, this could be stated "For all expressions e and all values v_1 and v_2 such that…".)
However, we will *not* prove this conjecture, because I don't think there's a short proof. (At least, not a short, easily generalized proof. There will be a process of "updating" proofs as we expand our language, and I'd like that process to go as smoothly as possible.) The problem is that, while I am confident that we will always get the same results v_1 and v_2, the rules give us the freedom to choose which part of the expression to step first, second, etc. (For the first step alone, I think we have $O(k)$ choices, where k is the size of e.)
We now have options:

(1) Change our rules so that they are both deterministic "in the end" *and* deterministic "along the way".

(2) Introduce a different flavour of semantics that will make it easier to prove determinism.

Eventually we'll pursue option (1), but I want to show you the new flavour now anyway (it gives an easier proof, and it will allow us to explore a connection to natural deduction).
The stepping judgment we have defined is called *small-step*; the new flavour is *big-step*. We only need two rules:

$\boxed{e \Downarrow v}$ expression e evaluates to value v (big-step)

$$\frac{}{n \Downarrow n}\ \text{eval-const} \qquad \frac{e_1 \Downarrow n_1 \qquad e_2 \Downarrow n_2}{(+\ e_1\ e_2) \Downarrow n_1 + n_2}\ \text{eval-add}$$

Now we can state something that's easier to prove:

Conjecture 4.
For all e, v_1, v_2
such that $e \Downarrow v_1$ and $e \Downarrow v_2$, it is the case that $v_1 = v_2$.

Proof. By structural induction on e.
…wait, what? □

What is structural induction?

4 Induction

Proofs in programming languages rely heavily on induction, often structural induction, because the objects being studied are *defined* inductively (recursively). Our grammar for e can be read as an inductive (recursive) definition:

(a) If n is an integer, then n is an expression.

(b) If e_1 and e_2 are expressions, then `(+` e_1 e_2`)` is an expression.

That describes how expressions are constructed. If someone else has constructed an expression, then we are not interested in how to construct the expression but how to "destruct" it: if we look inside an expression, we may find smaller expressions (subexpressions).

Likewise, rules constitute an inductive (recursive) definition. Our big-step rules, for example, can be read as the following:

(eval-const) If n is an integer, then

$$\frac{}{n \Downarrow n}\,\text{eval-const}$$

is a derivation.

(eval-add) If $\mathcal{D}_1$ is a derivation of $e_1 \Downarrow n_1$ and $\mathcal{D}_2$ is a derivation of $e_2 \Downarrow n_2$, then

$$\frac{\begin{matrix}\mathcal{D}_1 \\ e_1 \Downarrow n_1\end{matrix} \qquad \begin{matrix}\mathcal{D}_2 \\ e_2 \Downarrow n_2\end{matrix}}{(+\ e_1\ e_2) \Downarrow n_1 + n_2}\,\text{eval-add}$$

is a derivation. (New notation! I labelled the derivations of $e_1 \Downarrow n_1$ and $e_2 \Downarrow n_2$ by writing the derivation names above them.)

Remark . The similarity between these "expanded" definitions—of expressions on one hand, and big-step evaluation on the other—suggests that we could define syntax (like expressions) using rules instead of BNF grammars. But I prefer to use grammars for syntax, because then I know that I am only defining syntax—I am not defining addition, functions, or anything else about *computation*, only the shapes of expressions (or statements, procedures, programs). Grammars are also more compact.

Since every expression is constructed according to the inductive definition of expressions, which has two parts, we can prove something about *all* expressions by proving it for two cases. Suppose e is an expression.

- First case: The expression e was constructed using part (a).

  In this case, we know that e is an integer n: the definition said, "If n is an integer".

- Second case: The expression e was constructed using part (b).

  In this case, we know that e has the form `(+` e_1 e_2`)` where e_1 and e_2 are expressions.

The second case is where induction becomes essential. In an inductive proof, we get to assume what we are trying to prove for *smaller problems*. Proving something for the expression n seems like a smaller problem than proving it for a large expression.

Suppose we want to prove that, for all expressions e, the number of left parentheses LP in e is equal to the number of right parentheses RP in e:

For all expressions e, we have $\mathrm{LP}(e) = \mathrm{RP}(e)$.

The size of the problem must be related to e—the only thing we have is e. We will say that the problem for an expression e_S (Small) is smaller than the problem for an expression e_B (Big) when e_S is a *proper subexpression* of e_B. By analogy with proper subsets, e_S is a proper subexpression of e_B if e_S is a subexpression of e_B and $e_S \neq e_B$.

We will sometimes write $e_S \prec e_B$ to mean that e_S is a proper subexpression of e_B.

The assumption that our result—the conjecture we want to prove—holds for smaller problems is called the inductive assumption, or *inductive hypothesis* (IH).

To find the inductive hypothesis, we begin with the statement of the conjecture:

For all expressions e, we have $\mathsf{LP}(e) = \mathsf{RP}(e)$.

This is *not* our inductive hypothesis—if it were, we could prove anything! Instead, we must restrict the statement to expressions that are *smaller* than the given expression e.

This is a two-step process. Step 1 is renaming: to talk about whether something is smaller than e, we can't call the something e. So we rename e to e_S.

For all expressions e_S, we have $\mathsf{LP}(e_S) = \mathsf{RP}(e_S)$.

However, this is just as excessively strong as before, because it would let us choose e as our e_S and prove anything. We need to add a condition restricting e_S:

Induction hypothesis: *For all expressions e_S such that $e_S \prec e$, we have* $\mathsf{LP}(e_S) = \mathsf{RP}(e_S)$.

Conjecture 5. *For all expressions e, we have* $\mathsf{LP}(e) = \mathsf{RP}(e)$.

Proof. **By structural induction on e.** (Equivalently: By induction on the structure of e.)

- First case: The expression e was constructed using part (a). Therefore, e is an integer n. Integers do not contain parentheses, so $\mathsf{LP}(n) = \mathsf{RP}(n) = 0$. Since $n = e$, we have $\mathsf{LP}(e) = \mathsf{RP}(e)$.

- Second case: The expression e was constructed using part (b).

  In this case, we know that $e = (+\ e_1\ e_2)$ where e_1 and e_2 are expressions.

  The expressions e_1 and e_2 are proper subexpressions of e. Therefore, e_1 is smaller than e, and e_2 is smaller than e.

  By induction hypothesis on e_1,
  $$\mathsf{LP}(e_1) = \mathsf{RP}(e_1)$$
  That is, with e_1 as our e_S.

  By induction hypothesis on e_2,
  $$\mathsf{LP}(e_2) = \mathsf{RP}(e_2)$$
  That is, with e_2 as our e_S.

  Since $e = (+\ e_1\ e_2)$, we have
  $$\begin{aligned}\mathsf{LP}(e) &= 1 + \mathsf{LP}(e_1) + \mathsf{LP}(e_2)\\ \mathsf{RP}(e) &= \phantom{1 + {}}\mathsf{RP}(e_1) + \mathsf{RP}(e_2) + 1\end{aligned}$$

  We need to show $\mathsf{LP}(e) = \mathsf{RP}(e)$.
  $$\begin{aligned}\mathsf{LP}(e) &= 1 + \mathsf{LP}(e_1) + \mathsf{LP}(e_2) &&\\ &= 1 + \mathsf{RP}(e_1) + \mathsf{RP}(e_2) && \text{By above equations}\\ &= \mathsf{RP}(e_1) + \mathsf{RP}(e_2) + 1 && \text{Rearranging terms}\\ &= \mathsf{RP}(e) && \text{Above equation}\end{aligned}$$
  □

Unlike some proof techniques (such as case analysis—reasoning by cases), proof by induction demands commitment: you need to say, at the beginning of the proof, that you are doing a proof by induction *and what kind of induction you are using*. In the above proof, we did structural induction on e. In later proofs, we may have two expressions—are we doing induction on the first or the second? (Or both—there are several kinds of induction on multiple things. I will introduce those kinds as needed.) Or we may have derivations, as well as expressions.

Some authors like to present structural induction with a restriction: the induction hypothesis only works for *immediate* subexpressions. That is, if k "layers" of the inductive definition were needed to create $(+\ e_1\ e_2)$, then $k-1$ layers were needed to create e_1 and e_2. Our proof above would survive that restriction, since we do only use the IH on the immediate subexpressions e_1 and e_2. In general, however, we sometimes want to use the IH on "sub-sub-expressions". Every proof that's valid using restricted structural induction is valid using complete structural induction, so I always use the complete kind.

(This corresponds to the distinction between simple and complete induction on the natural numbers. For a problem on a natural number k, in simple induction the IH is the goal for $k-1$; in complete induction, the IH is the goal for all $k' < k$. Since $k-1$ is indeed less than k, every proof that uses simple induction remains valid using complete induction, but not the other way around.)

References

Gerhard Gentzen. Untersuchungen über das logische Schließen. *Mathematische Zeitschrift*, 39: 176–210, 405–431, 1934. English translation, *Investigations into logical deduction*, in M. Szabo, editor, *Collected papers of Gerhard Gentzen* (North-Holland, 1969), pages 68–131.