CPSC 311: Operational semantics **(DRAFT)** ("lec-operational")

Joshua Dunfield
University of British Columbia

September 16, 2016

What do programs mean? They mean whatever the[1] language definition says they do. So the real question is: How do we specify, in a language definition, the meaning of the language's programs?

Remark. I'm starting to use the lecture notes from 2015, updating as I go. In addition to outright mistakes, there may be references that are out of date, such as notes that something was done in lecture—meaning a 2015 lecture. I'll try to fix these before lecture, but I will miss some; please point them out (a note on Piazza is a good method).

1 Logistics

- **Start on the assignment!**
  We can tell how many people have run `handin`.
- The assignment is due 1 day **before** the drop deadline.
- If you can't finish the assignment, you should probably drop the course.
- Run `handin` early and often!
  (Even if you haven't done a single problem yet!)

2 Dynamic semantics

Dynamic semantics is about **how** programs behave:

- Dynamic semantics tells you how to "step" a program.
- Or how to "evaluate" a program.
- These methods work a little differently, but they have the same purpose: they tell you what your interpreter is supposed to do.

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

Dynamic semantics is about **how** programs behave:

[1] We'll assume that we know *which* language the program is written in, despite programs such as `http://ideology.com.au/polyglot/polyglot.txt`.

- Dynamic semantics tells you how to "step" a program.
- Or how to "evaluate" a program.
- These methods work a little differently, but they have the same purpose: they tell you what your interpreter is supposed to do.
- "Interpreter semantics":
  "to explain a language, write an interpreter for it." . . . "When we finally have what we think is the correct representation of a language's meaning. . . "
- If the interpreter you write defines the language, you **cannot** know whether it's correct. (It's trivially correct, because it defines itself. "When the President does it, that means it is not illegal.")
- You can test it on programs, but tests can only show the presence of bugs, not their absence!

3 Why dynamic semantics?

Unlike syntax, where practically all language designers[2] uses some variation of BNF grammars, specifying which syntactically well-formed programs actually mean something, and what they mean, is less settled.

Various methods have been used, with names like "axiomatic semantics", "operational semantics", "natural semantics", and "denotational semantics". Within the programming languages research community, there is lively competition amongst these methods. To most of the world, though, this competition is off the radar: most languages' semantics are specified informally. (Standard ML is probably the most popular formally defined language—and Standard ML is even less "mainstream" than Scheme/Racket.)

In 311, we will focus on one method, *operational semantics*. Given a specification in operational semantics, it is relatively easy (compared to specifications using other methods) to write an interpreter. Operational semantics has a rich mathematical foundation, which you would want to understand to do research in programming languages, but you don't need to understand that foundation to turn operational semantics into interpreters.

The idea of trying to specify the meaning of a mathematical object (a program) through natural language alone, rather than more "formally" (through logic and mathematics), calls to mind a quotation:

> "About the use of language: it is impossible to sharpen a pencil with a blunt axe. It is equally vain to try to do it with ten blunt axes instead." —Edsger Dijkstra

Formal mathematical language is not an absolute guarantee against mistakes or oversights in defining the semantics (a serious mistake in SML's formal definition went unnoticed for years), but it, at least, gives us a point of reference. Rules in natural language are for people, not computers; understanding a programming language shouldn't require one to be a "language lawyer".

[2] One exception: the designers of Algol 68, who tried to innovate in this area; it didn't end well.

4 Evaluation semantics

As I mentioned, operational semantics is closer to an interpreter than other methods for specifying what a program does. There are different flavours of operational semantics; we'll start with the one that's usually easier to understand, called *evaluation semantics*.

(That is, evaluation semantics is one kind of operational semantics, which is one method for specifying dynamic semantics. But for now, just remember: what we're going to do, right now, is called evaluation semantics.)

The idea of evaluation semantics is that the dynamic behaviour—the dynamic "meaning" of a program—is *the value it computes*, or equivalently, *what it evaluates to*. We expect that {+ 2 2} will compute 4, or equivalently, will evaluate to 4.

For our very first example of evaluation semantics, we'll follow the language "AE" of arithmetic expressions (from Chapter 2 of PLAI). Its concrete syntax (BNF grammar) is:

$$\begin{aligned}\langle \texttt{AE} \rangle \ ::= \ & \langle \texttt{num} \rangle \\ & \mid \{+ \ \langle \texttt{AE} \rangle \ \langle \texttt{AE} \rangle\} \\ & \mid \{- \ \langle \texttt{AE} \rangle \ \langle \texttt{AE} \rangle\}\end{aligned}$$

And its abstract syntax (PLAI, p. 6) is:

```
(define-type AE
 [Num (n number?)]
 [Add (lhs AE?) (rhs AE?)]
 [Sub (lhs AE?) (rhs AE?)])
```

However, we'll use the concrete syntax to write the evaluation semantics. This allows programmers to use our semantics to understand the language, provided they can read evaluation semantics rules. The only people who should have to know what the abstract syntax looks like are the people writing the interpreter.

Now, let's write down a specification of the dynamic behaviour of this language, using the method of evaluation semantics. First, we should ask: what is our goal? How will we know when we have a complete (not necessarily *good*, but complete) specification? One answer (which is not always the right answer, but will work just fine for this language) is: if we can specify the meaning of all expressions that are syntactically well-formed (according to the EBNF for $\langle \texttt{AE} \rangle$), then we have a complete specification.

Thus, we need to specify the meaning of each of the three syntactic cases ($\langle \texttt{num} \rangle$, $+$ and $-$) in the EBNF. Since we're going to use evaluation semantics, we need to specify what a $\langle \texttt{num} \rangle$ evaluates to, what a $+$ evaluates to, and what a $-$ evaluates to.

This language is so tiny that there's only one reasonable way it can work: $+$ should add, and $-$ should subtract. So we can focus on *how* to write down an evaluation semantics, rather than spend time wondering if we're making good design decisions.

We want to be *really* precise, so let's try to be a little more precise than just saying "$+$ should add, and $-$ should subtract". Just as CPSC 110 shows how to follow a data definition (for example, if you need to write a function that takes a BST (binary search tree), you need to write a case for 'false' and a case for '(make-node ...)'), let's try to follow the BNF:

- we need to say what a number evaluates to,

- we need to say what a $+$ evaluates to, and
- we need to say what a $-$ evaluates to.

This is a little vague, though, because we didn't mention the subexpressions of $+$ and $-$. Let's fix that, and also (in the first case) mention the specific number!

- we need to say what a number n evaluates to,
- we need to say what $\{+\ ae1\ ae2\}$ evaluates to, and
- we need to say what $\{-\ ae1\ ae2\}$ evaluates to.

Here, $ae1$ stands for the first subexpression, and $ae2$ stands for the second subexpression.

Now let's actually say (in English) what these things evaluate to. A number shouldn't *do* anything, so we'll say that it evaluates to itself:

- A number n evaluates to n.

What should $\{+\ ae1\ ae2\}$ evaluate to? Well, that depends on what $ae1$ and $ae2$ are. Or rather, what they evaluate to. So let's start there.

- If $ae1$ evaluates to n_1, and $ae2$ evaluates to n_2, then $\{+\ ae1\ ae2\}$ evaluates to ...

We want $+$ to *add*, so it needs to add n_1 to n_2.

- If $ae1$ evaluates to n_1, and $ae2$ evaluates to n_2, then $\{+\ ae1\ ae2\}$ evaluates to $n_1 + n_2$.

Now we can give meaning to $-$ in the same way, resulting in something reasonably precise (it's still in English):

- A number n evaluates to n.
- If $ae1$ evaluates to n_1, and $ae2$ evaluates to n_2, then $\{+\ ae1\ ae2\}$ evaluates to $n_1 + n_2$.
- If $ae1$ evaluates to n_1, and $ae2$ evaluates to n_2, then $\{-\ ae1\ ae2\}$ evaluates to $n_1 - n_2$.

4.1 Rules

We're now very close to an evaluation semantics! In fact, all we have to do is rewrite the above using some funny notation: Instead of "AE evaluates to n", we'll write "$AE \Downarrow n$". And instead of "If... then ...", we'll use a horizontal line, like this:

$$\frac{ae1 \Downarrow n_1 \qquad ae2 \Downarrow n_2}{\{-\ ae1\ ae2\} \Downarrow n_1 - n_2}\ \text{Eval-sub}$$

An *inference rule*, or *rule* for short, looks like

$$\frac{\text{premise}_1 \qquad \ldots \qquad \text{premise}_m}{\text{conclusion}}\ \text{rule name}$$

The part below the line is called the *conclusion*, and the parts above the line are called the *premises*. To the right of the line, we often write the name of the rule. We can read a rule as follows: To derive the conclusion, we must satisfy each of the premises. In other words, if the premises are satisfied, we have shown the conclusion. Or, very briefly, "if premises, then conclusion".

This notation was invented in the 1930s by the logician Gerhard Gentzen, for the purpose of formally expressing mathematical proofs. At the moment, we're using this notation to talk about what evaluation means in a (very small) programming language, but the notation is much more general. The logical statements that appear as conclusions are called *judgments*.

4.1.1 When is a judgment derivable?

A judgment is *derivable* if we can get it by applying rules. You may want to think of "derivable" as meaning "true"; this is okay, as long as you keep in mind that "truth" is entirely a matter of what rules you have.

4.1.2 Premises and conclusions

Rules always have a conclusion, but they don't have to have premises. In fact, to write down the rule for our first case ("A number n evaluates to n"), we don't need any premises:

$$\frac{}{n \Downarrow n}\ \text{Eval-num}$$

Often, the evaluation rule for a syntactic form will have exactly one premise for each smaller expression it contains. A number doesn't contain any subexpressions, so the evaluation rule for numbers doesn't have any premises. On the other hand, {− ae1 ae2} has two subexpressions, so its rule has two premises.

4.1.3 Applying rules

What can you do with a rule? You can *apply* it, by filling in its "meta-variables". Here, our "meta-variables" are n (in Eval-num), and ae1, ae2, n_1, and n_2 in Eval-add and Eval-sub. A meta-variable is a placeholder: we can fill in ae1 and ae2 with ⟨AE⟩'s, and we can fill in n, n_1 and n_2 with numbers.

This is easier to see with an example. Given the rule

$$\frac{}{n \Downarrow n}\ \text{Eval-num}$$

we can apply it by plugging in an actual number for the meta-variable n:

$$\frac{}{7 \Downarrow 7}$$

Once we've applied Eval-num, we have an *evaluation derivation* of $7 \Downarrow 7$, and say that we have *derived* $7 \Downarrow 7$.

Note that, unlike our EBNF grammar—where we wrote ⟨AE⟩ twice in the production for + to refer to (possibly) *different* expressions—writing n twice in the rule Eval-num means that we have to substitute the same number.

We can similarly derive $6 \Downarrow 6$:

$$\frac{}{6 \Downarrow 6}$$

This gives us two derivations, one of $7 \Downarrow 7$ and one of $6 \Downarrow 6$, so we have enough derivations to apply Eval-sub:

$$\frac{\dfrac{}{7 \Downarrow 7} \qquad \dfrac{}{6 \Downarrow 6}}{\{- \; 7 \; 6\} \Downarrow 1}$$

We got this by looking at the rule Eval-sub, plugging in 7 for $ae1$, plugging in 6 for $ae2$, plugging in 7 for n_1, and 6 for n_2. The conclusion of Eval-sub says "$\ldots \Downarrow n_1 - n_2$", which—after plugging in for n_1 and n_2—is $\ldots \Downarrow 7 - 6$, which is $\ldots \Downarrow 1$.

Notice that this derivation of $\{- \; 7 \; 6\} \Downarrow 1$ looks like a tree (oriented the natural way, with the root at the bottom, rather than the usual computer science way). And in fact, derivations are also called derivation trees. This is a nice feature of Gentzen's notation: derivations "fit together" visually.

Here's a slightly larger example:

$$\frac{\dfrac{\dfrac{}{20 \Downarrow 20} \qquad \dfrac{}{2 \Downarrow 2}}{\{+ \; 20 \; 2\} \Downarrow 22} \qquad \dfrac{\dfrac{}{7 \Downarrow 7} \qquad \dfrac{}{6 \Downarrow 6}}{\{- \; 7 \; 6\} \Downarrow 1}}{\{- \; \{+ \; 20 \; 2\} \; \{- \; 7 \; 6\}\} \Downarrow 21}$$

It's often useful to write the names of the rules being applied (later languages will have more than just three rules!):

$$\dfrac{\dfrac{\dfrac{}{20 \Downarrow 20}\,\text{Eval-num} \qquad \dfrac{}{2 \Downarrow 2}\,\text{Eval-num}}{\{+ \; 20 \; 2\} \Downarrow 22}\,\text{Eval-add} \qquad \dfrac{\dfrac{}{7 \Downarrow 7}\,\text{Eval-num} \qquad \dfrac{}{6 \Downarrow 6}\,\text{Eval-num}}{\{- \; 7 \; 6\} \Downarrow 1}\,\text{Eval-sub}}{\{- \; \{+ \; 20 \; 2\} \; \{- \; 7 \; 6\}\} \Downarrow 21}\,\text{Eval-sub}$$

4.2 Evaluation rules for AEs

In PL research papers, it's customary to collect all the evaluation rules together, and throw one giant figure at the reader. Fortunately, we only have three rules.

$$\frac{}{n \Downarrow n}\,\text{Eval-num} \qquad \frac{ae1 \Downarrow n_1 \qquad ae2 \Downarrow n_2}{\{+ \; ae1 \; ae2\} \Downarrow n_1 + n_2}\,\text{Eval-add} \qquad \frac{ae1 \Downarrow n_1 \qquad ae2 \Downarrow n_2}{\{- \; ae1 \; ae2\} \Downarrow n_1 - n_2}\,\text{Eval-sub}$$

The section "When is a judgment derivable?", above, may help with the following exercises.

Exercise 1. Using the above rules, is $\{* \; 1 \; 2\} \Downarrow 3$ derivable?
How about $\{* \; 1 \; 2\} \Downarrow 2$?

Exercise 2. Suppose we added a rule

$$\frac{ae1 \Downarrow n_1 \qquad ae2 \Downarrow n_2}{\{* \; ae1 \; ae2\} \Downarrow 2} \text{ Eval-mult}$$

Now, using the 3 rules above *and* Eval-mult, is $\{* \; 1 \; 2\} \Downarrow 2$ derivable?

5 From the rules to an interpreter

Now we'll write an interpreter that follows our evaluation rules. This interpreter will turn out to do the same thing as PLAI's interpreter from Chapter 2. (**2016 note:** Don't worry if you haven't read that chapter.) The difference is how we got there. Once you understand how to write interpreters based on evaluation rules, you can take evaluation rules you've never seen before—and that may define a language with features you've never heard of—and write an interpreter that follows those rules.

You won't get that skill instantly just from this one tiny language, but you have to start somewhere! And it's easier to start here than with something like *The Definition of Standard ML*.

5.1 Restating the rules in abstract syntax

It's easier to work with abstract syntax—the "AE" defined with define-type—than concrete syntax, so our interpreter will accept programs in abstract syntax. You can learn to mentally translate between concrete and abstract syntax, but for now, let's explicitly translate the rules to abstract syntax. We just have to change all the AEs, inserting the constructors Num, Add and Sub.

$$\frac{}{(\text{Num } n) \Downarrow n} \text{ Eval-num} \qquad \frac{ae1 \Downarrow n_1 \qquad ae2 \Downarrow n_2}{(\text{Add } ae1 \; ae2) \Downarrow n_1 + n_2} \text{ Eval-add} \qquad \frac{ae1 \Downarrow n_1 \qquad ae2 \Downarrow n_2}{(\text{Sub } ae1 \; ae2) \Downarrow n_1 - n_2} \text{ Eval-sub}$$

This shows something interesting, though: the animals on each side of the "evaluates to" arrow ($\Downarrow$) are not the same kind of animal.[3] In Eval-num, we have an AE, $(\text{Num } n)$, on the left of $\Downarrow$, but a plain number n on the right. In the concrete syntax, we didn't write Num explicitly, so we couldn't see this difference. We could have chosen, instead, to "evaluate cats to cats" and produce an AE on the right, but it's a little more convenient to produce a number. (Later in 311, we'll define other flavours of operational semantics that don't work this way.)

The job of writing an interpreter for AEs boils down to writing a function that answers this question:

"Given an ae, find a number n such that $ae \Downarrow n$."

During lecture, we wrote the following function:

```
(define (interp ae)
  (type-case AE ae
    [Num (n) n]
    [Add (ae1 ae2)
```

[3] In honour of my undergrad discrete math professor's advice: "You must always ask yourself: what kind of an animal is it?"

```
      (let ([n1 (interp ae1)]
            [n2 (interp ae2)])
        (+ n1 n2))]
  [Sub (ae1 ae2)
      (let ([n1 (interp ae1)]
            [n2 (interp ae2)])
        (- n1 n2))]
  ))
```

Our `interp` function behaves the same as the `calc` function in PLAI, but our function has more let-bindings. This is more verbose, but strengthens the connection between our interpreter and the rules. For example, the expression `(+ n1 n2)` is a direct Racket translation (parentheses and a prefix operator +) of the $n_1 + n_2$ that appears in the conclusion of Eval-add.