Operational semantics questions and answers

COMP 105

27 January 2020

Contents

Fundamentals

Why is the syntax / value distinction important?

The most helpful answer I can think of is in metaphor: Syntax is a letter saying you've been admitted to Tufts. Values are what Dowling produces when they evaluate your admission letter: you get a place in the freshman cohort, the ability to enroll in classes, and so on.

Judgments

What are some examples of judgments? Everything is so abstract right now.

Examples:

- $v = 7$
- $x \in \text{dom } \rho$
- $n = m$
- $\langle \mathit{LITERAL}(99), \xi, \phi, \rho \rangle \Downarrow \langle 99, \xi, \phi, \rho \rangle$
- $\langle \text{APPLY}(*, \text{VAR}(y), \text{LITERAL}(3), \xi, \phi, \{y \mapsto 7\} \rangle \Downarrow \langle 27, \xi, \phi, \{y \mapsto 7\} \rangle$
- $\langle e, \xi, \phi, \rho \rangle \Downarrow \langle v, \xi', \phi, \rho' \rangle$

What is the most basic meaning of a "judgment"?

It is a thing that could be true—or to be completely technically correct, it is a thing that might be *provable*.[1] In addition to the examples above, here are some more judgments, written in informal English:

- v is seven
- The value of x is seven.
- x isn't defined.
- Class is held even in the event of snow.
- Two plus two is four.
- Two plus two is five (not true or provable).

What about the $\Downarrow$ symbol?

That's a notation that separates initial state (with syntax) from final state (with a value) in the evaluation judgment for *expressions*. The evaluation judgment for definitions has a different form and is written with the $\rightarrow$ arrow.

In what contexts do we actually apply the big-step judgment form? Is it in derivation trees?

Yes. And also when we're describing the result of an evaluation.

Rules

Is a rule simply a premise and conclusion that include judgments?

Yes, separated by a horizontal line. A *useful* rule is this, plus it describes something about actual computation. Here's an example of a useless rule: "whenever global variable `foo` is defined, the expression always evaluates to zero." This rule describes a language that nobody would want.

Could you please clarify the notation for the Impcore rules?

If above the line, all the evaluations described by judgments happen, and all the conditions in the other judgments are satisfied, then the evaluation below the line happens.

What is the difference between the $\Downarrow$ on the top and bottom of the ~~judgment~~ rule? What do they each represent?

Both represent the evaluation of an expression. In the Impcore semantics, the one below the line represents the evaluation of an expression. A $\Downarrow$ above the line represents the evaluation of a *subexpression*—like the right-hand side of an assignment or the condition in a conditional expression.

What is the difference between a judgment and a rule?

A language's forms of judgment determine what you can *say* about evaluation. In Impcore, we can say only what happens when we evaluate an expression and what happens when we evaluate a definition. A language's rules determine what's *true* about evaluation. (We can say two plus two is

[1] The distinction between truth and provability is deeply important to mathematical logic and to the foundations of computer science, but we will try not to sweat it.

five, but it isn't true, and there is no combination of rules that can prove it.)

Here's another difference: we typically have just one form of judgment per *syntactic category*: in our case, one for definitions and one for expressions. But we have at least one rule for each *syntactic form*: one for VAL, one for DEFINE, one for literals, two for variables, two for SET, two for IF, two for WHILE, one for empty BEGIN, one for nonempty BEGIN, and so on. For each syntactic form, we need at least one rule, or else there is no way to evaluate the form.

How should we think about / interpret inference rules?

They say what happens when we run the code.

Environments and their notation

What does the notation $\{x_1 \mapsto n_1, \ldots, x_k \mapsto n_k\}$ *mean?*

It's an environment that defines k variables. The names of those variables are x_1 through x_k, and for any i in the range 1 to k, the value of variable x_i is n_i.

What does a double prime signify?

It signifies a name that might be different from a similar name that has only one prime, or no primes, or three or more primes.

Is $\rho'' = (\rho')'$ *or* $(\rho)''$ *?*

None of the above. Priming is not an operation in the sense that, say, squaring a number is. We write ρ, ρ', and ρ'' in order to get three different names that all name similar kinds of things—in this case, formal-parameter environments. Just like the names of program variables, these names are technically arbitrary; they are chosen primarily to make sense to a reader. So if I thought that formal-parameter environments would be suitably named for 20th-century comedians, I could name my environments Groucho, Harpo, and Chico. If I preferred 21st-century novelists, I could name them Leckie, Jemisin, and Kowal.

Please don't do this on your homework.

What is the difference between subscripts, primes, and double primes in the context of the ~~APPLY function~~ ApplyUser *rule shown in class?*

They are all different names for similar objects. For example, both ξ_0 and ξ_2 describe states that hold the values of all the global variables. And because they have different names, they might be different—we don't know.

Whenever there is an e in your big-step judgment, is it likely that you will see a prime on the right-hand side of $\Downarrow$*?*

Great question! Yes. In particular, if a rule has an e on the bottom, we are practically guaranteed to pick up a prime in the final state. In any context, even a derivation, if we are evaluating an *unknown* e, we're going to get primes. But watch out for some of the homework derivations! When all the parts of an expression are known, we don't get primes (because there's no unknown e).

How do we know when globals vs formals are changing in a function call?

Both could change. To know which variable is changed by a SET, we look at the three environments in the initial state in which the body of the function is evaluated.

Impcore operational semantics

What's the exact difference between x*,* e*, and* v*? How are they related to each other?*

They are all *metavariables*. x names a program variable, e names an expression, and v names a value. You'll find much more information in the book, in section 1.1.2 on page 15.

What is v *in the big-step judgment form?*

It's the result of evaluating the expression e.

In terms of environments, how is $\rho\{x \mapsto v\}$ *used and applied in judgments?*

It's used exclusively when a formal parameter is assigned to: in the FormalAssign rule. You'll see the very similar $\xi\{x \mapsto v\}$ when a global variable is assigned to: in the GlobalAssign rule.

When ~~declaring~~ defining a variable, is there still a premise such as $x \in dom\ \xi$ *or* $x \in dom\ \rho$*?*

Not in Impcore. Look at the DefineGlobal rule on page 35.

Why can every change be put into the buckets ξ *and* ρ*? Is that Impcore-specific, or an overarching rule?*

Because ξ and ρ stand for the data structures that hold the values of all the variables. There's no place else for values to go. That's an Impcore-specific design.

Where do e_1 *and* e_2 *come from in the* ApplyUser *derivation?*

They are the actual parameters to function f, and they are part of the syntax of the function-call expression.

So we have three environments in Impcore. Where are actual parameters stored? Do they go into the global variables category?

Good question! Actual parameters are stored in the formal-parameter environment. This sounds screwy until you realize that a formal parameter is simply a mechanism for naming the value of an actual parameter. So when we call a user-defined function, the body of that function is evaluated in formal-parameter environment $\{x_1 \mapsto v_1, \ldots, x_k \mapsto v_k\}$,

and in that environment values v_1 to v_k are the values of the actual parameters.

In the ~~APPLY function~~ ApplyUser *rule, the bottom ρ_0 evaluates to ρ_2. Why? Is that from the premises, or does ρ_2 represent another value?*

It might be easier to understand if think of the formal parameters as a privileged species of local variable, and you pronounce it this way: "variables ρ_0 are changed to ρ_2." And it's from the premises: first we evaluate e_1, and that changes variables to ρ_1. Then we evaluate e_2, and that changes variables to ρ_2. Finally, we evaluate e, but it is in another function and doesn't have access to our variables. So when e is done, we still have ρ_2.

I would like to understand more in depth when certain symbols change (ξ, ϕ, ρ, etc.).

ξ and ρ change when a `set` expression is evaluated. ξ also changes when a `val` definition is evaluated. And ϕ changes when a `define` definition is evaluated.

How do we determine which environment(s) changed while evaluating an expression?

The rules tell you. And if we study the rules, we can conclude that environment change is driven by syntax: ξ changes if you SET a global variable, or if you call a function that sets a global variable. And ρ changes if you SET a formal parameter.

Do global functions go in ξ or ϕ?

Impcore has only one form of function, and they all go in ϕ.

What does $\phi(f) = \text{USER}(\langle x_1, \ldots, x_n \rangle, e)$ mean?

"`f` is a user-defined function with n formal parameters named x_1 through x_n and with body e." (This statement captures everything we need to know about an Impcore function in order to run it—everything that's in its definition.)

In the Literal *rule, do we have any constraint on v being an actual number?*

Not built into the rule. To ensure that v is a representable number, we rely on the parser. To see what happens if v can't be represented, try entering the numeral written with ten 9's.

What are the types we need to know? Ex: VAR, LITERAL, …

All of the syntactic forms of expressions and of true definitions.

Derivations

Proof techniques in past math courses I've taken sometimes seem to require the development of intuition in knowing when to pull something out of nowhere in order to prove something. Will this occur for our derivation proofs at any point? Right now it seems to be working solely backwards, which is nice.

Good question! The answer is no. Derivations can be constructed mechanically; intuition would only get in the way. Derivations could easily be constructed by a computer program; for example, it is a tedious but straightforward exercise to adapt the interpreter in Chapter 1 so it produces derivations. (Tedious mostly because representing derivations in C is a pain.)

How do you approach starting a derivation?

Great question! The algorithm is described in the book on pages 57 and 58, but the gist of it is, look at the expression in the initial state, and find the one or two rules which have the form of that expression in the conclusion. Proceed from there.

How does the derivation rule apply to the rules for operational semantics we saw in the handout?

The rules in the handout are the rules for *constructing* derivations.

How does recursion factor into derivation trees? How do we represent recursive functions with derivation trees? Prove them?

The derivation tree will contain an ApplyUser node, and one of its subtrees will contain another ApplyUser node with the same function. As long as a recursion terminates, there are absolutely no issues writing a derivation for it—the process for a recursive call works the same as for any other call.

How would you do derivations on recursive function calls, and can you give an example?

Same as for regular calls, and I can't fit an example in this space. I've asked the TAs if they can come up with an example.

Could you clarify how recursion impacts ρ?

Great question! Every activation of the recursive function gets its own private ρ, which doesn't interfere with any of the others. This property, which was one of the great innovations of the late 1950s, is implicit in the ApplyUser rule. But to puzzle it out requires careful thinking.

When working on a derivation that uses one of the assigment rules, can we define shorthand for modified environments and put this shorthand above the line?

Yes and no. Yes, it is extremely useful to define abbreviations for modified environments. No, you probably don't want to bury your abbreviations in the middle of your derivation. Put your abbreviations off to the side in a table, where they can easily be seen.

Can you give us more information on / examples of derivations?

Yes. There's more info and an example in the book, and you'll do another in recitation.

How do you know what ξ and ρ get evaluated to (whether primed, double primed, or the same) when writing derivations, especially in more complex ones?

Take it one step at a time and follow the rules of the semantics.

How do you construct a derivation tree where a premise contains more than one judgment?

Write subderivations side by side. Have a look in the book on pages 55 and 56.

How does one perform a valid substitution in a derivation?

Great question! Each node must be obtained from a rule by *consistent substitution*. For example in the LITERAL rule, if I substitute 99 for v on the left of the $\Downarrow$, I must also substitute 99 for v on the right. Or in the IFTRUE rule, if I substitute `(> n 0)` for e_1 below the line, I must also substitute `(> n 0)` for e_1 above the line—and therefore provide a subderivation that shows what `(> n 0)` evaluates to.

How deeply nested should derivations get?

Really deep. Roughly as deep as the number of times each function is called, times the size of that function's body, summed over all active functions.

In implementation, do we need to keep track of the three environments for every single state?

Yes. But it's easier than it sounds:

- When a new global-variable environment ξ' is computed, the old one is thrown away. So in practice, a data structure or machine state can be updated in place.
- Tracking ϕ is even easier, as it changes only when a definition made with `define` is evaluated. Old ones are thrown away.
- There could be multiple formal-parameter environments active at once, but there is only one per each active function, and they can all be stored on a stack. (This is a big part of the call stack that you may have studied in COMP 40.)

Can you provide examples of evaluation judgments that involve mixing all the concepts (global variables, function applications, assignment, etc.)?

Holy cats! If you provide me with *syntax* that mixes all the concepts, I'll write the evaluation judgment. Post it to Piazza.

What does it mean for primes to never be in a derivation?

It means that every environment in the derivation is either equal to an initial environment or is formed from a previous environment by an update operation (like $\rho\{x \mapsto v\}$).

Applications

How is all this applicable to programming?

It's most applicable to diagnosing and debugging code. If you're looking at code—or a language manual—and you're not quite sure what the code is doing (at a low level), the operational semantics has the answers.

Can you tell a well-designed language based on its derivation tree?

No. The derivation tree tells you more about the *computation* than about the language. Good design is more likely to involve *properties* of derivation trees, like "given initial environments, each expression has at most one valid derivation," or "there is no derivation in which high-security information influences the value of a low-security variable."

I would like to further understand how derivations and operational semantics are used in coding and the real world.

Think of derivations, rules, and judgments in the same way you would think about assembly language and machine architecture: they tell you how the machine works. They operate at too low a level to be useful thinking tools.

- In coding, they are used to resolve tricky corner cases or to answer language questions. Like, for example, "if I add 1 to x but x isn't defined, what happens?" Answers might be different in, say, Awk, Scheme, Python, or JavaScript.
- The most powerful use of derivations is in *metatheory* (next lecture), which gives properties of whole languages, like the security property in the previous question.

The real world uses these tools when the code really has to be right—like in nuclear power plants and airplanes. When the code doesn't have to right but just has to be delivered, using these tools is less likely. Although you will still find them in pockets of places like Google and Mozilla.

Where do these rules and judgments fit in when we think about the 9-step design process?

Rules and judgments justify algebraic laws.

What is the best way to think of this theory while writing implementations?

As a debugging tool. If your code has a bug, you first boil it down to a failing unit test. If you can't solve it then, you keep making smaller unit tests, until your failing unit test is so small that maybe you've misunderstood how some syntactic form is supposed to work. At that point you consult the operational semantics.

How can I apply this structure to understanding a language?

Great question! I barely have room to touch on this question, but here are a couple of examples:

- Operational semantics really helps narrow down the design space of what's sensible. For example, from class, if there is any name x, either $x \in \text{dom } \rho$ or it isn't, and either $x \in \text{dom } \xi$ or it isn't. So any time syntax names a variable, there at most four sensible ways for it to behave, and you can identify them all. Operational semantics teaches you to ask the question.
- As another example, operational semantics can identify exactly what is meant by a concept like "truthiness" in JavaScript. It's the condition corresponding to the judgment $v \neq 0$ in the IfTrue rule. Change the judgment and you change what "truthiness" means. More importantly, you understand why there is even such a concept as truthiness—it's to give meaning to `if`.

  Where else do we see that judgment? Only in the WhileIterate rule. And sure enough, that's also about truthiness. (If your `while` and `if` have different notions of what's true, then your language design sucks.)

Sketching out operational-semantics rules is a really good way to look at a language you've never seen before and to figure out how much is really new and how much is just old wine in new bottles.

Language design and implementation

Do there always need to be exactly three environments to make a basis, or is that just the case in Impcore?

Good question! That's just Impcore. Over the course of the next six weeks, we'll eventually see two other examples of bases.

How do judgments translate to a programming language if they define the language itself?

I would say that the judgments define a language and that they translate to, among other things, an implementation of that language. The most direct form of translation is called a *definitional interpreter*, and you will find one for Impcore in the book in section 1.5.

Can you implement a language with only the summary of its operational semantics?

Yes. That's quite a valuable exercise. In March, when you have more experience under your belt, you'll do something related, which is to implement a type checker with only the summary of its rules.

How are rules formed (e.g. filling in the Literal *rule in class; can we systematically do that?)*

Rules capture our understanding of what happens when we run the code. As an experienced CS student, you already have a pretty solid understanding of these things—you just need to learn a new way of writing down that understanding. You'll have a chance on the homework, on exercise 16, parts (a) and (b).

How can I successfully make new proof systems? What are the best ways to think about this?

Viciously difficult question. I would start out with a very clear and detailed mental model of running code (or implementation) and then start to define that using a proof system. From that starting point, there are a lot of directions in which you can branch out. Come see me in office hours.[2]

Examples from class

What does $\langle 10, ... \rangle$ *in the derivation rules slides mean?*

Good question! It's a final state in which I've written only the value $v = 10$, and to make the example fit on the slide, I've left out all the environments.

The `(* (+ 10 1) (- 10 1))` *example but with the detailed semantics from the handout would have been nice to see as they are much more confusing (the structure of inference rules is confusing, too.)*

Agreed. That derivation would be really big, but you might be satisfied with the one on page 56 of the textbook, which is quite similar.

For the Literal *rule, I'm still confused why we don't need* $x \notin \xi$.

Look at the syntactic form of Literal—there is no name, so there is no x.

It's still unclear why in the GlobalVar *rule we need both* $x \notin \text{dom } \rho$ *and* $x \in \text{dom } \xi$ *but the same doesn't apply to the* FormalVar *rule.*

It's a language-design choice, and its designed to allow the following program:

```
(val n 99)
(define example (n)
   n)

(example 33)
```

We want this program to run and for the call to `(example 33)` to return 33. If you put the extra condition on the FormalVar rule, the program above won't run, because no condition for evaluating VAR(`n`) will be satisfied.

[2] It's worth 5% of your grade.

General class suggestions and questions

For the upcoming assignments, do we only count ~~real~~ "true" definitions as definitions, or are we including expressions as well?

I'm not sure what you mean by "count" here. Maybe clarify your question on Piazza?

I would appreciate a dictionary of all the terms we should know right now, with examples.

This is a 100-level course. Part of "taking responsibility for your own learning" is that if you need a glossary, you prepare it. ("Glossary" is the term of art for a dictionary of technical terms with explanations or examples.) The glossary at the end of the chapter might provide a useful starting point. Beyond that, keep your ears open. Identify terms you don't know. Figure out which ones are important. Organize your learning.

You may as well start now, while the stakes are low, because when you get your first job, you'll be expected to do this sort of thing for yourself.

Do we need to study the textbook, or will the lectures be sufficient to understand stuff?

As I mentioned at the beginning of the second lecture, lecture is just the tip of the iceberg. In lecture, I try to point out what's ahead and highlight the most difficult bits, or do things where interaction could really help learning. You definitely need to study the textbook. But don't study all of it! For most situations, it should suffice to study the parts mentioned in the homework assignments.

Miscellany

Why do we like recursion more than loops?

Great question! Because a recursive function has a name and a contract, and it can easily be unit tested. A loop has no name and no contract, and it's much harder to write tests for.

Favorite Birkenstock sillhouette?

Arizona.

What's your haircare routine?

Lather. Rinse. Repeat.[3]

[3]Why did the computer scientist starve to death in the shower?