Operational semantics questions and answers

COMP 105

30 January 2019

Contents

Functions vs syntactic forms

What makes something a syntactic form? And how do you know when you have them all?

Something is a syntactic form if and only if it is so specified in the language's grammar. In Impcore, μScheme, and μSmalltalk, almost every syntactic form is identified by a "reserved" word (like `val`, `define`, `if`, `set`, `while`, `begin`) following an open bracket (round or square). The reserved words can be found in typewriter font in the book section on "Concrete syntax."

Environments and their notation

What do the different environment notation variations mean?

Any variation, as in prime or subscript, is a way of notating similar environments that might be different. For example, ξ and ξ' both notate global variables, but they might notate different values for the same global variables, or even different sets of variables.

What does the state of ρ look like for a function "a" called within a function "b"? How are their ρ's related?

They are not related. Each function gets its own ρ which is constructed from the names of its formal parameters and the values of its actual parameters.

Does changing the value of a variable already declared in the global environment modify ξ?

Notionally, it produces a new ξ' that differs from the preceding ξ only in that one variable. (But a decent implementation will implement this new environment by updating a data structure.)

What is the difference between $\rho\{x \mapsto 3\}$ and $\rho'\{x \mapsto 3\}$? Are they the same? Is one correct?

They notate the same operation ("set x to 3") undertaken at two different starting points (ρ and ρ'). They are the same if and only if ρ and ρ' are the same.

Do we always have ρ' and ξ' after making a function call?

You would see notation ρ' only if the effect of evaluating the actual parameters is unknown. And you would see ξ' only if evaluating the actual parameters has an unknown effect on global variables, or if indeed the call itself has an unknown effect on global variables. If the parameters and functions are known, you'll have known effects, and the final environments will be formed from ρ and ξ using updates.

Function environments

Does ρ only contain the name-value pairs of formal parameters in the function currently being evaluated or is it for all potential function calls?

Just the function currently evaluated. On the hardware, each function gets its own stack frame, and in the semantics, each function gets its own ρ.

If ϕ never changes in evaluations of expressions, how are any new functions defined?

With a definition form: the `define` form.

Are there any languages where the function environment ϕ changes?

Yes, but in my experience, such a language treats a function just like any other value, and the semantics combines ξ and ϕ. We will see in our next semantics that it is possible (and useful!) to combine all three environments (ξ, ϕ, and ρ) into one.

Is operational semantics used a lot in compiler or just in theory?

There is some experimental work using special tools to build a compiler or interpreter directly from an operational semantics. But the classic compilers are based on an older style of semantics called *denotational* semantics. Such compilers are very common; in the compiler world that technique usually goes under the name "syntax-directed translation."

Understanding and writing derivations

Why are some premises above the judgment line next to each other versus on top of each other?

The placement of premises is formally meaningless. But to improve readability, we try to write them in an order that is consistent with the order of execution. And if multiple premises will fit on one line without compromising readability, we will do that to save vertical space.

In a derivation tree, if we want to show that $\rho(x) = 100$, is it correct to display a judgment as $\langle e, \xi, \phi, \rho\{x \mapsto 100\}\rangle \Downarrow \langle ... \rangle$, or is it better to include $\rho(x) = 100$ as a premise?

Which one is correct depends on what you want to say:

- Saying $\rho(x) = 100$ means that we know a fact about ρ, namely that x is *initially* 100.
- Writing $\rho\{x \mapsto 100\}$ says that you are starting the computation in a state that is like ρ, except x is 100.

If you are given a fact about ρ, the first one is correct—the second would be misleading.

Can we see some examples of even bigger derivation trees? I think it would help contextualize what is meant by the induction on "smaller" derivations.

I cannot fit a big example here, and they are tiresome to typeset. But if you come to my office hours (or any TA), we can create a big derivation tree on a whiteboard.

What is a derivation? [Shows a mock derivation tree with $\mathcal{D}_r$ above the line] is $\mathcal{D}_r$ the premise?

A derivation is a static data structure or proof that represents or describes a computation. In the notation used in class, $\mathcal{D}_r$ is the name of a derivation. (In the example in class, it is the name of a subderivation of a premise.)

What does $\mathcal{D}_r$ imply? Is it a way of labeling or does it represent arbitrary expressions that allow us to conclude the induction hypothesis?

It is a way of labeling an arbitrary *valid* derivation. So it can't be arbitrary *expressions*; it has to be expressions whose evaluation terminates in the given state.

Recursive calls of derivations in assumption, e.g., while.

No problem! The recursive derivation is smaller, because the `while` loop terminates in fewer steps.

When dealing with named variables like x in $\mathrm{SET}(x, 3)$*, do you replace it with* $\mathrm{VAR}(x)$ *when doing a derivation?*

No. The VAR tag identifies an *expression* form, and the left-hand side of a SET is always a name. Writing, e.g., `SET(VAR(y), e)` would be ill-formed syntax (not a well-formed expression).

Applications of operational semantics

How do we know what rule applies for a given statement or piece of syntax of the language? Why does the syntax of the language match the rule 1:1 sometimes and other times not?

A rule applies if you can substitute for all its metavariables (e, ξ, and so on) to obtain the judgment. A good first step is to match the syntax in the conclusion, below the line. The syntax matches the rules 1:1 in those cases where there is only one possible way a syntactic form can be evaluated (e.g., LITERAL). It does not match when there is more than one thing that can happen (e.g., IF can be true or false, and it changes the evaluation).

Why are we learning about operational semantics? Why are operational semantics necessary?

You're learning about operational semantics for two reasons: to start to think precisely about programming languages, and to be able to communicate with other educated people. A semantics is necessary only if you need to answer the question "what is this code supposed to do?", as opposed to the approach, "let's fire up the implementation and see what happens this week."

Why *operational* semantics? It is the easiest flavor to learn, and it comes with powerful tools for proving so-called "safety" properties, like "no well-typed program leaks private information."

Do we need to know all syntax before doing operational semantics?

To write a rule of operational semantics, you need to know what syntactic form you are writing about. But you can start to write a semantics (or start on the homework) without knowing every single syntactic form.

How do I convert operational semantics into actual code and will we be doing that in C?

In actual code, you do case analysis on the syntactic form, then you run computations on the premises, until eventually you know what rule applies and what to do. We will not be doing that in C, and in fact we will not be doing that this term. Instead we will do a very similar task, namely translate type systems into actual code. If you want a preview, you can look at the fifth lesson on program design.

How do I deal with different cases over the same syntax, like if, when which rule applies is based on an unknown?

The rule that applies is based on a premise above the line. In a proof, if you truly don't know which premises can be proved, then you have to consider all possible cases. In each one, you simply assume that the rule applies, and go on to show what you need to show. In an implementation, you write code to discover a provable premise. Like testing $v \neq 0$ or calling function `isvalbound` to test $x \in \operatorname{dom} \rho$.

Why is there a different notation for definitions in operational semantics when definitions can be expressions? It seems like def $\supseteq$ exp, but somehow the operational semantics of expressions seems more complex.

Definitions do more than expressions: they add new names to ξ and ϕ. It is reasonable for the expressions to seem complex because there are more syntactic forms and many more rules. But if you look at the forms of the evaluation judgment, you may conclude that neither one is more complex than the other; they are just different.

How do we express in operational semantics that `val` *and* `define` *are not expressions and cannot be in expressions in impcore?*

That is syntactic structure, not run-time behavior. In other words, your statement answers the question "how can I write a well-formed program," not "what happens when I run the code." Syntactic structure is expressed not by operational semantics but by a grammar. In the case of Impcore, this would be the grammar on page 6.

How do I choose rules in uncertain situations?

By seeing which premises, if any, are satisfied.

Doing proofs and metatheory

Are there any tips for identifying major components of proofs or how to get started with a proof?

Figure out which rules can be used in a derivation. (Given known syntax, just some of them. Given unknown syntax, all of them.) Then just pick a case and dive in. The book section on "How to attempt a metatheoretic proof" is pretty good.

When we are proving $\operatorname{dom} \xi = \operatorname{dom} \xi'$*, we assume that it holds for base cases; how do we know what base assumptions we can use?*

You don't assume it for base cases—you have to prove it for every case, base case or inductive case. In metatheory, there are typically no base assumptions you can use. What we *can* assume is that when we are given a *smaller* subderivation, $\operatorname{dom} \xi = \operatorname{dom} \xi'$ holds because the induction hypothesis holds. The ability to use the induction hypothesis on any smaller object is the very essence of proof by induction. (In math class, you may have heard this technique called "strong" induction.)

How do you check your answer after doing a structural induction proof?

Two recommendations:

- Try substituting known examples for unknown expressions, environments, and so on, and make sure what you're claiming actually works out.
- Anywhere you've applied an induction hypothesis, make sure you've applied it correctly. Especially this: if the induction hypothesis is an implication, you must show that the precondition is met, or you won't be able to rely on the conclusion.

Is there a simple algorithm to solve an induction proof that doesn't rely on having a "feeling" for what to do?

The only part that might rely on "feelings" is coming up with the right induction hypothesis. So the algorithm is this: try an induction hypothesis that's identical to what you wish to prove, and dive in. Keep an eagle eye out for cases where the induction hypothesis doesn't quite work. If you find one, look for a stronger induction hypothesis (and for that last step, feelings do help).

Why are we using induction to prove things?

It's the only way to prove facts about infinite sets, such as all programs or all terminating computations.

Can you assume that $\operatorname{dom} \xi$ *will not change. Or in impcore is there a way it can?*

I have *proved* that the evaluation of an *expression* cannot change $\operatorname{dom} \xi$. I was not able to assume that; I had to prove it.

How do you know when to use the induction hypothesis?

Any time you have a thing that is smaller (e.g., a smaller derivation), that is a candidate for the induction hypothesis.

Do you always look at the same rules/base cases?

You always look at the same rules. And the base cases are always the same as well.

Why do we do this complicated "Tony Hawk" inductive proof setup instead of just doing it linearly like is taught in math class/set theory?

Because derivations are trees, and trees are not linear. Trees have a strictly more complex structure than natural numbers.

How do you know all base cases are covered? It seems that given one faulty base case, in this system it's possible to trivially prove anything.

You are correct. You have to prove every single case, including all of the base cases. And yes, if you get one case wrong, the whole proof is wrong. (Base cases are no more or no less susceptible of being wrong than any other case.)

New language designs

For different languages with different features, are their evaluation forms different and do you just add more Greek letters? What happens when you run out of Greek letters?

Yes, different languages have different judgment forms for their evaluation. We tend to reuse the same Greek letters over and over, within reason. If you run out of Greek letters, your design is probably too complicated, but if that doesn't deter you, you can use Roman letters, Hebrew letters, and Fraktur, and you can decorate letters with hats, tildes, bars, arrows, and more. The ingenuity of mathematicians knows no bounds.

Design of semantics

Why are the IfTrue and IfFalse rules separate in the handout semantics?

Because an `if` expression *behaves* differently depending on whether the condition is zero or nonzero.

What's involved in creating your own rule (e.g., giving Impcore Awk-like ~~syntax~~ semantics)? Do you just make a rule and it applies in every case? Or do you make a rule and have to account for it throughout the previously established rules as well?

When we say "creating your own rule," we mean "adding a new rule to an established system while respecting existing rules and judgment forms." A new rule can often coexist peacefully with existing rules, but sometimes it might replace one or more existing rules. If you want to change the form of the evaluation judgment, which you might do to account for local variables, for example, then you have to revise all the existing rules to support the new judgment form.

When creating local variables, do we make this as an expression or definition?

In Impcore, neither. Local variables are specified as part of the definition form for functions. Here's an example:

```
(define nthprime (n)
   [locals k]
   (begin
      (set k 2)
      (while (!= n 1)
         ...)))
```

How does one approach searching judgments and derivations for error or overlooked cases?

Error cases are tricky, because in an operational semantics, error cases are cases in which there is no proof. So they are not represented explicitly. You would start by trying an example. If you want to prove that a case leads to error, such as proving that `(+ n 1)` leads to an error when `n` is not defined, you would have to do it by contradiction: assume there is a derivation, then show that the assumption leads to a contradiction. In the example, you might show that a derivation is incompatible with the assumption that `n` is in neither ξ nor ρ.

Examples from class

How much structural induction must be justified? e.g., in the IfTrue case, must we show both $\mathcal{D}_1$ and $\mathcal{D}_2$?

The induction hypothesis is there for *both* $\mathcal{D}_1$ and $\mathcal{D}_2$, so if you have to use it, it's already paid for (because both $\mathcal{D}_1$ and $\mathcal{D}_2$ are strictly smaller derivations). But you're not required to use it; there are certainly some theorems for which it is not necessary to apply the induction hypothesis at every single opportunity.

If $\operatorname{dom}\xi = \operatorname{dom}\xi'$, we know ξ and ξ' have the same variables, but nothing is said about the values. Would the notation $\xi = \xi'$ suggest that ξ and ξ' contain the same global variables with the same values?

Exactly right.

Could you (theoretically) write out $\mathcal{D}_r$ all the way? For the example we did in class on 1/30.

Yes. You are *guaranteed* that this is possible, because every valid derivation is finite.

What does "by induction hypothesis on $\mathcal{D}_r$" mean? Does it have to do with literals?

It means that I can assume that because $\mathcal{D}_r$ is a valid, smaller, derivation, I am guaranteed that its initial ξ and its final ξ' have the same domain. It has nothing to do with literals.

[Presumably in a derivation of something like an expression with If] Why are $\mathcal{D}_1$ and $\mathcal{D}_2$ not the same?

Because $\mathcal{D}_1$ describes the evaluation of the condition e_1, whereas $\mathcal{D}_2$ describes the evaluation of the "true branch" e_2, and these are not necessarily the same.

Is a rule just one of the things on the sheet from last class?

Yes.

Metacognition

How do we bridge the gap between following along in lecture and writing proofs on our own?

Recitation will help, as will the book section "How to attempt a metatheoretic proof."

How do I get to a comfortable amount of knowledge where I know what I don't know and can ask a question?

I'm having trouble understanding this question—maybe the transcription is wrong? I won't hold up the rest of the answers, but I'll try to track down the original card and see what it says.

What's the best way to practice and reinforce the jargon and symbols of operational semantics?

Examples! Particularly examples where you translate to code and back. Here's one: how do you say, "the result of evaluating `(+ n 1)` is independent of the value of `x`"? Once you can say a few things like this using the jargon and symbols of operational semantics, you'll be all set.

For those of us who haven't taken Comp 160/Comp 170 and haven't seen discrete for 2 years, how best can we prepare for what's to come, because this all seems scary and frustrating?

Keep two ideas in mind:

- Induction is just recursion dressed up in mathematical clothes.
- At bottom, this is all about code. You know code.

Class got Jokes

ρ, ρ, ρ your boat, gently down the ξ

My hovercraft is full of eels.

What do you get when you cross a narwhal with a person and place it in a small section of the ocean with some rams?

A Nar-man Ram-sea.