Notation

Does the notation always show first the initial step, then intermediate, then final?

An evaluation judgment below the line (in a conclusion) always shows the initial state to the left of the \( \downarrow \) and the final state to the right. The number of intermediate states depends on whether evaluation judgments appear above the line, as premises.

What does \( D \) represent?

It represents a syntactic proof of an evaluation judgment. The \( D \) stands for derivation, which is the name for this kind of syntactic proof.

Why is \( D \) smaller than overall derivation?

Because it’s a proof of a judgment above the line, so building the full derivation requires adding one more line than is found in \( D \). (It’s the same reason that in a nonempty binary tree, the left child is smaller than the overall tree.)

Do we have to specify what \( D \) is each time or can we just write \( D \)?

You normally use \( D \) (or \( D_1 \) or \( D_2 \)) to name a derivation, in much the same way that you use \( e \) or \( e_1 \) to \( e_2 \) to name an expression. Typically you don’t have to know about the form of \( D \); you just have to know that it satisfies the induction hypothesis.

Where do we get the judgment form for evaluation? I.e., where do we explicitly state that evaluation doesn’t change \( \phi \)?

As the language designer, I picked the judgment form. It’s discussed explicitly in the book on page 19.

What do inference rules represent?

Each inference rule represents one piece of the answer to the question “what is the code supposed to do at run time?” Collectively, the inference rules explain how every valid program is supposed to behave.

The underlying concepts of opsem seem (somewhat) simple; why do we complicate them with such esoteric symbols and syntax?

For much the same reasons that we program computers using esoteric symbols and syntax—the notation serves a special purpose. Nobody starts out finding \( \langle v, \xi, \phi, \rho \rangle \downarrow \langle v, \xi', \phi, \rho' \rangle \) easy to read, any more than anybody starts out finding

```c
#define FLAG(P,F) ((P)?(Bitpack_ ## F):0)
```

easy to read. But it will grow on you.

In the case of operational semantics, the notation provides a powerful, dense tool for language definition. For a combination of precision, information density, and abstraction, nothing can beat it. Figures 1.4 and 1.5 on pages 77 and 78 define an entire programming language. Defining that same language in C requires over two thousand lines of code.

I’m unsure about the structure of proofs. When I see

\[
\begin{array}{c}
E & F \\
\hline
C & D \\
A \downarrow B
\end{array}
\]

my understanding is that we’re proving from \( A \) to \( B \) with intermediate steps \( C, E, F, D \) in that order.

Your understanding is as good as far as it goes, but it’s incomplete; the layout you show is conventional, but the layout doesn’t determine order. Order is determined by examining initial and final states. Your diagram might represent this typical sequence of events:

- The initial state \( A \) is the same as the initial state of \( C \).
- The initial state of \( C \) is the same as the initial state of \( E \).
- The final state of \( E \) is used to build the final state of \( C \).
- The final state of \( C \) is the initial state of \( D \).
- The initial state of \( D \) is the same as the initial state of \( F \).
- The final state of \( F \) is used to build the final state of \( D \).
- The final state of \( D \) is used to build the final state \( B \).

Details that cover all proofs are a little more complicated, but this explanation will do to go on with.

Can we do a quick blast through the different notations involved in the data structure?

Happy to do it at office hours.

Evaluation doesn’t add or remove a global

In the cases in the proof for \( \text{var}(x) \), if it were true that \( x \in \text{dom} \rho \), could we still say that \( \text{dom} \xi = \text{dom} \xi' \)?

Yes! If there is a derivation in that case, it has the form
\[ x \in \text{dom } \rho \]

\[
\frac{\text{var}(x), \xi, \phi, \rho}{\downarrow (\rho(x), \xi, \phi, \rho)}
\]

Here the final global environment is identical to the initial one, so our proof obligation is \( \text{dom } \xi = \text{dom } \xi' \), which is true.

If instead we have rule FormalAssign, we’re still good, but in that case we get \( \text{dom } \xi = \text{dom } \xi' \) from the induction hypothesis.

**Why is it important that the domain of \( \xi \) stay the same?**

It’s not important. But it’s a good example of something that is easy to prove about Impcore, and that is not necessarily true in every language, because designers of other languages have made other choices.

**Why does global Assign require that \( x \) is already in \( \text{dom } \xi \)? Can we not initialize new global variables?**

In Impcore, the only way to introduce a new global variable is with a \text{val} definition. It’s that way because in my role as language designer, I thought requiring an explicit definition would be better software-engineering practice—you can identify all the global variables with a quick scan through the definitions.

**What’s the point of having a \( \xi \) and \( \xi' \) variable in these operational semantics for the global variables if we know the \( \xi \) doesn’t change?**

But \( \xi \) might change! All we know is that the \text{domain} of \( \xi \) doesn’t change. (In the judgment form, the prime on \( \xi' \) means that \( \xi \) and \( \xi \) might differ, as discussed in the book on page 22.) For example, if \( n \) is a global variable, then evaluating \( \text{set } n \ (\ + \ n \ 1) \) changes \( \xi \), but it doesn’t change \( \xi \)’s domain.

**Induction**

I am having a lot of trouble understanding the inductive step. It seems like nothing was ever proved, and we rely on it already being proved?

It’s just like recursion. In a recursive function, it’s not true that “nothing was ever implemented”—even though the function can call itself. Eventually the recursion bottoms out in code that doesn’t make a recursive call.

It’s the same way in a proof. Eventually the proof “bottoms out” in a base case, which is proved without the help of the induction hypothesis.

**What, in a derivation, is the induction performed on?**

Strictly speaking, the structure of the derivation. But most students are more comfortable thinking about “strong” induction over the height of the derivation tree.

**Why are we allowed to assume that \( \text{D} \) implies that \( \xi \) doesn’t change?**

That’s how inductive proofs work. We’re allowed to assume an induction hypothesis whenever derivation \( \text{D} \) is strictly smaller than the thing we’re trying to prove. It works because a tree is an inductive structure—and we can prove all the cases, including the base cases where there is no \( \text{D} \) above the line.

**How do we know our inductive hypothesis is correct when we are assuming it in the metatheoretic proof?**

We know it is correct only if (a) we are assuming it about a smaller derivation and (b) we can prove all the cases which might be used to create a derivation—including the base cases.

**For a complete metatheoretical proof, do we have to mention the base cases with literals and variable evaluations? Or do we just assume that they are a given?**

You must absolutely mention these cases. In fact, you must do more: for each base case, you must prove the induction hypothesis for the judgment below the line. In these kinds of proofs, base cases are super important. If you ignore the base cases, you can “prove” nonsense like “evaluating any expression produces delicious cheese.” That’s easy to prove for all the inductive rules (the compound syntactic forms), but you will find yourself absolutely unable to prove the base cases. For example, there is no way to prove that evaluating a literal \( 7 \) produces delicious cheese.

**What if a base case changes the domain of \( \xi \), so that \( \text{dom } \xi \neq \text{dom } \xi' \)? Is that even possible?**

It’s possible only if you extend or change the language. For example, I could extend Impcore with a \text{forget} expression, which I would define with this rule:

\[
\frac{\text{forget}, \xi, \phi, \rho}{\downarrow \langle 99, \{ \}, \phi, \rho \rangle}
\]

The rule says that evaluating \text{forget} returns \( 99 \), and as a side effect, it destroys all the global variables, leaving \( \xi \) empty.

For your homework, you’ll change Impcore so that set can enlarge the domain of \( \xi \), leaving \( \text{dom } \xi \subseteq \text{dom } \xi' \). (You’ll change just the semantics, not the implementation.)

**Language-design alternatives**

Also, when we prove that something is true, how is it possible that we can design a language for which it is not true?

When you design a language, you make up your own inference rules for the evaluation judgment. The proof we did in class relies on the rules for evaluation of Impcore, and if there is any change in the rules, no matter how minute, our proof could be invalidated.
**Nontermination**

Are there ways of reasoning built into our operational semantics/meta-theory to help reason about evaluations that we don’t know terminate?

No. The big weakness of the big-step semantics is that it can’t say anything about a nonterminating computation.

How do we write proofs about expressions that might not terminate or might not have a valid derivation?

Using big-step semantics, you can’t. For these kinds of proofs you need a small-step semantics. We’ll see a small-step semantics in March, but we won’t do any proofs with it. (The small-step proofs are harder.)

**Changing functions**

What languages, if any, allow dynamic changes to $\phi$?

Off the top of my head, examples include Scheme, Perl, Python, and Lua. But of these examples, only Perl has a separate $\phi$. In the other languages, a function is just like any other value, and it lives in $\rho$.

**Evaluating definitions**

Do the operational semantics cover what happens after a single definition is evaluated?

The operational semantics in section 1.4.7 on page 26 say how to evaluate a single definition, but not what happens afterward. In practice, what happens after you evaluate a single definition is that you evaluate the next definition, and so on until you run out of definitions. But I chose not to specify that formally.

The operational semantics do say what the final state is after evaluating a definition: it’s a new basis comprising just $\xi'$ and $\phi'$.

**Function application**

Can you go through APPLYUSER again? Don’t quite understand the flow of the derivation.

Happy to do it in office hours.

Why is “ApplyUser” not called “ApplyFunction”? Why not?

Because you can also apply primitive functions, and each primitive function has its own rule or rules. Have a look at the examples on page 25.

**Applications of formal semantics**

Where are we going with this? I love it, and I think having some sort of end goal or next step would focus and connect my learning.

Our goal in 105 is to master the notation so that we can use it to talk about how programming languages work and what different language features do. One of the big outcomes of the course is becoming fluent in reading and writing inference rules. In the big leagues, they go beyond reading and writing to write interesting proofs. Quite often the proofs are to help justify an implementation technique or to make sure we can trust the code about which something is proved.

If this a standard proof technique in functional programming, how are proofs verified for complicated languages? It seems infeasible to verify all these proofs by hand in the peer-review process.

It is a standard proof technique not just for functional programming languages but for all programming languages. And because a derivation is a data structure, it is not hard to write a program to check to see if any given derivation is valid. Such a program is called a “proof checker,” and by the end of this term, you will have developed skills that would enable you to write a proof checker easily, in perhaps a week or so.

Professionally, the most popular tool for creating and validating formal proofs is the Coq proof assistant [1]. The easiest way to get started with Coq and with machine-checked proof is with the online book *Software Foundation* [2] by Benjamin Pierce et al.

How can I use derivation to my advantage when reasoning about general problems in code? E.g. would it be useful in making sure I cover all my cases?

I don’t think derivations can help make sure you cover all cases. Derivations are actually a pretty low-level tool, and for reasoning about code you are better off with something like algebraic reasoning (which we study in 105) or axiomatic semantics and refinement (which we don’t study in 105, but you could consult work of C.A.R. Hoare). For proofs of correctness of imperative code, the best tool is probably predicate-transformer semantics, which you could learn about in *The Science of Programming* by David Gries (recommended for beginners) or *A Discipline of Programming* by Edsger Dijkstra (not recommended for beginners).

**Harmless animals at risk**

Why do cats die when somebody asks about strong vs. weak typing?

I have discovered a truly marvelous answer to this question, which this handout is too small to contain.

---

1. [https://coq.inria.fr/](https://coq.inria.fr/)
2. [https://www.cis.upenn.edu/~bcpierce/sf/current/index.html](https://www.cis.upenn.edu/~bcpierce/sf/current/index.html)