Assignment: Operational Semantics

CS 105

Due Tuesday, September 20, 2022 at 11:59PM

Contents

If you're going to talk about languages you've never seen before, you need a vocabulary. This assignment introduces you to the basics of operational semantics, inference rules, and syntactic proof technique. You will use these skills heavily throughout the first two-thirds of the course, and after 105 is over, when you want to understand a new language idea, you will use them again.

Some of the essential skills are

- Understanding what judgment forms mean, how to read them, and how to write them
- Understanding what constitutes a valid syntactic proof, known as a *derivation*
- Understanding how a valid derivation in the operational semantics relates to a successful, *terminating* evaluation of an expression

- Proving facts about families of programs by reasoning about derivations, a technique known as *metatheory*
- Using operational semantics to express language features and language-design ideas
- Connecting operational semantics with informal English explanations of language features and program behaviors

Few of these skills can be mastered in a single assignment. When you've completed the assignment, I hope you will feel confident of your knowledge of exactly the way judgment forms, inference rules, and derivations are written. On the other skills, you'll have made a start.

On this assignment, **there is no pair programming**.

> The most unusual problem on this assignment is *metatheory*. What's the role of metatheory? If somebody is trying to sell you a language you have never seen before, they might try to sell it on the basis of some kind of guarantee. For example, in the Singularity project, Microsoft tried to sell the language "Sing#" on security and reliability grounds: that any program written in "Sing#" would meet their reliability claims. If you know metatheory, you'll know whether to buy what somebody is selling about "any program."

Retrieval practice

Before tackling reading-comprehension or other homework questions, you have the option to check your understanding by looking at the "retrieval practice" questions in section~1.10.1, which starts on page~75 of the textbook. Look at questions G through L, and N. (Example question: what does the metavariable x stand for?) If you can answer these questions quickly and easily, then you are prepared to tackle the actual reading-comprehension and other homework questions. If you can't answer a retrieval-practice question, a TA will be happy to answer it for you.

Do not submit answers to any retrieval-practice questions. They are indicators of preparedness, nothing more.

Reading comprehension

Before starting the programming problems, answer the reading-comprehension questions for this module, which you will find online. It is also OK to alternate between reading-comprehension questions and related homework questions.

Part A: Talking operational semantics (25%)

The homework is divided into parts A, B, and C. This assignment is almost exclusively a theory assignment, and almost everything goes into one file `theory.pdf`. (The exception is some code you will write for part (d) of exercise~16, in part B, which goes into file `awk-icon.imp`.) In this part (part A), you *translate* between colloquial, informal English and the language of operational semantics.

Related reading:

- For rules of operational semantics, see section~1.5, which starts on page~29. The most important rules are summarized on pages~80–81.

Translating operational semantics into English. From exercise~10 on page~79 of *Build, Prove, and Compare*, complete parts (a) and (d). Your informal English must not use any symbols that refer to environments—talk about environments using your knowledge of what names stand for. Make your explanations as simple as possible, and *use only language that a beginning programmer (e.g., a CS 11 student) would understand.* (In other words, it is OK to talk about "variables," but it is not OK to talk about "environments.")

Translating English into operational semantics. From exercise~11 on page~79 of *Build, Prove, and Compare*, complete parts (a), (c), and (d). Notes:

- The convention of the field is that new metavariables are *implicitly universally quantified*. For example, if you just write a global-variable environment ξ, you're assumed to be talking about *any possible* ξ. If what you're saying is true for just *some* environments, the conventional opening is to say "there exists a ξ such that." (Or more likely, "There exist ξ, ϕ, ρ, v, ξ', and ρ' such that.")
- If you need to write an implication, use the words "if" and "then." Do *not* use inference-rule notation—every new inference rule risks inadvertently changing the language definition, and it adds an additional obligation (a new case) to every metatheoretic proof.
- Some of the statements in this problem most easily formalized in the form "if *evaluation judgment* then *conclusion*."

Exercise R. *Meanings of rules.* The following notation suggests an alternative to the WhileEnd rule of Impcore. Either explain why this rule is effectively the same as the original rule, or explain why it's different.

```
        ⟨e1,ξ,φ,ρ⟩ ⇓ ⟨0,ξ′,φ,ρ′⟩
--------------------------------------- (WhileEnd')
⟨while(e1, e2), ξ, φ, ρ⟩ ⇓ ⟨0, ξ′, φ, ρ′⟩
```

The following notation suggests an alternative to the FormalAssign rule of Impcore. Either explain why this rule is effectively the same as the original rule, or explain why it's different.

```
  x ∈ dom ρ    ⟨e,ξ,φ,ρ⟩ ⇓ ⟨v,ξ′,φ,ρ′⟩
------------------------------------------------ (FormalAssign')
⟨set(x, e), ξ, φ, ρ⟩ ⇓ ⟨v, ξ′, φ, ρ′⟩
```

Note: A convincing explanation of why two rules are different is should include not only an explanation in English but also a code example on which the two rules behave differently.

Part B: Operational semantics and language design (25%)

This part contains just one exercise, in which you explore two alternative ways of dealing with unbound variables, and you write code to distinguish them.

We encourage you to discuss ideas, but *no other student may see your rules or your code*. If you have difficulty, find a TA, who can help you work a couple of similar problems.

Specifying new language features with new rules. Do all parts of exercise 16 on page~82 of *Build, Prove, and Compare*. This is an exercise in language design. The exercise will give you a feel for the kinds of choices a language designer might have made in a language you have never seen before. It will

also give you a tool you can use to think about the consequences of language-design choices *even without an implementation*.

To complete the exercise, you must analyze three variations on a design: the Impcore standard and two alternatives, which resemble the languages Awk and Icon. For the Impcore standard, you can confirm the results of your analysis using the Impcore implementation. But for the Awk-like and Icon-like variations, you don't have an implementation that you can use to verify the results of your analysis. To get the problem right, you have two choices: think carefully about the semantics you have designed and the program you have written—or build two more interpreters, so that you can actually test your code. (Each new interpreter requires only a two-line change to file `eval.c`, so if you wanted to build new interpreters, you wouldn't be deep in the weeds.)

Part (d) involves coding from scratch, and it could involve new functions. But these functions are not trying to do anything useful with data; instead, they are trying to tease out differences in language semantics. Moreover, unless you choose to build interpreters, you cannot run unit tests of the Awk-like and Icon-like semantics. For these reasons, the only steps we expect from our recommended design process are a name and a contract for each function you choose to write (steps 3 and 4).

We will assess your code by running it in three interpreters. This assessment leaves you vulnerable to these common mistakes:

- You might define a function and forget to call it. If you forget to call your function, then when we run your code, the last thing the interpreter does will probably *not* be to print 0 or 1, which is what is called for in the exercise.
- You might forget that after evaluating an expression, the interpreter prints the result of the expression.
- You might use `print` or `printu` where you really meant `println`.
- You might include unit tests in your code. In that case, the last thing the interpreter prints will be the results of running the unit tests.

Hint: For part (d), retrieval question I (page 75) is especially relevant.

Part C: Operational-semantics derivations, proofs, and metatheory (50%)

This part takes you into proof: theory, derivations, and metatheory. We encourage you to discuss ideas, but *no other student may see your rules, your derivations, or your proof.* If you have difficulty, find a TA, who can help you work a couple of similar problems.

Related reading:

- For an explanation of a valid derivation and for the algorithm used to build one, see section~1.7, which starts on page~56.
- For an example of a derivation tree, see page~58.
- For rules of operational semantics, see section~1.5, which starts on page~29. The most important rules are summarized on pages~80–81.
- For metatheory, see section~1.7.2, which starts on page~60.

Proof by derivation. Do exercise~12 on page~79 of *Build, Prove, and Compare*. The purpose of the exercise is to develop your understanding of derivations, so be sure to make your derivation *complete* and *formal*. You can write out a derivation like the ones in the book, as a single proof tree with a horizontal line over each node. If you prefer, you can write a sequence of judgments, number each judgment, and write a proof tree containing only the numbers of the judgments, which you will find easier to fit on the page.

If it helps to make your derivation fit on the page, you are welcome to use abbreviations like Lit for Literal, ForVar for FormalVar, GloVar for GlobalVar, and so on.

In this exercise, or in writing any derivation, the most common mistake made is to copy judgments blindly from the rules of the semantics. This kind of copying results in superfluous primes. In the rules, the primes in ξ' and ρ' are a way of saying "I don't know." In particular, what's unknown is the exact nature of the subexpressions, and therefore the results of evaluating them. (Notice that the syntactic forms Var and Literal don't have any subexpressions, and their rules don't have any primes.) In the expression `(begin (set x 3) x)`, all of the subexpressions are known, and a correct derivation doesn't have any primes.

Proof using derivations. Do part (a) of exercise~13 on page~79 of *Build, Prove, and Compare*. Now that you know how to *write* a derivation, in this exercise you start *reasoning* about derivations. This problem calls for a math-class proof *about* formal semantics, so any formal derivations you write need to be supplemented by a few words explaining what the formal derivation is and what role it plays in the proof.

As in the previous exercise, be wary of primes. The ξ', ρ', ξ'' and ρ'' in the problem are not necessarily different from the initial environments or from each other. The primes say only that they *might* be different.

Exercise F: Metatheory. This final exercise requires you to raise your game again, by reasoning about the *set* of all *valid* derivations. It's *metatheory*. Metatheoretic proofs are probably unfamiliar, but you will have a crack at them in lecture and in recitation.

It's an annoying quirk of the fall schedule, but metatheory won't be covered in lecture until the Monday before the homework is due. If you want to get started before then, Section 1.7.3 (page 62) has everything you need. The exercise you'll do is very similar to the final example on pages 63 to 66.

Metatheory is a fantastic tool, because it gives you results that can apply to *any* program written in a language. But it's hard to get started: there are useful metatheoretic results, and there are easy metatheoretic results, but I don't know any useful, easy metatheoretic results. Here are a couple of results that are useful but not easy:

- Impcore is deterministic: the same program gives the same answer every time (not true of Java!).
- Impcore can be evaluated using a call stack (like C and Java).

To make it possible to do something relatively interesting but also relatively easy, I'll ask you to investigate a metatheoretic conjecture that's not true:[1]

> **Conjecture**: If global variable y is defined, and if y does not appear in expression e, and if expression e evaluates to some value, then the evaluation of e does not change the value of y.

[1] A famous scientist once said that proofs in programming languages are interesting only when they are wrong.

To *disprove* a metatheoretic conjecture, it is sufficient to find a counterexample, but to show insight into how metatheory works, you will also explain what parts of the proof fail and how. To structure your explanations, you'll rely on a stylized structure that applies to every metatheoretic proof: as described in the book in section~1.7.3, a proof needs a case for each rule in the semantics.

Using the detailed definitions below, complete the following four tasks, each of which relates to the conjecture.

(a) Exhibit a counterexample. That counterexample will include an expression e whose evaluation changes the value of some variable y, even though y won't appear in e.

(b) Since the conjecture relates to a change in the value of a variable, the cases for assignment (`set`) are likely to be relevant. Both those cases actually work; show that they go through.

Note that the two cases are actually different; both need to be handled in full.

(To prove these cases, which are inductive, you will be proving an implication. Please identify the *induction hypothesis*, which is the left-hand side of that implication. You may wish to review the handout "Overview of Induction" by Chris Phifer.)

(c) Find one case of the metatheoretic proof that fails, and explain why the proof doesn't go through for that case.

(d) *Briefly* explain, in language a beginning programmer would understand, why the conjecture fails. (A sentence or two is plenty.)

To reduce bureaucracy, you will show that the conjecture fails in **Simplified Impcore**. Simplified Impcore is a restricted subset of Impcore in which:

- There are no `while` or `begin` expressions.
- Every function application has exactly two arguments.
- The only primitive function is `+`.

Using Simplified Impcore reduces the number of cases to something manageable.

Here's the conjecture in detail: whenever $\langle e, \xi, \phi, \rho \rangle \Downarrow \langle v, \xi', \phi, \rho' \rangle$ and y is a variable that is in dom ξ but **does not** appear ("free") in e, then $\xi'(y) = \xi(y)$. We will not formalize "appearing free" just yet, but the appearance of a free variable is defined inductively over the abstract syntax. Here is the definition for Impcore:

- x appears free in VAR(x).
- x appears free in SET(x, e).[2]
- If x appears free in any of e's subexpressions, then it also appears free in e. For example, if x appears free in e_1, then it also appears free in IF(e_1, e_2, e_3).

This concept is so central to the proof that it benefits from notation: for the set of variables that appear free in e, we write $\mathit{fv}(e)$. We can then make things more formal:

- $\mathit{fv}(\text{VAR}(x)) = \{x\}$.
- $\mathit{fv}(\text{SET}(x, e)) = \{x\} \cup \mathit{fv}(e)$.
- If e has subexpressions, its set of free variables is the union of the free variables of the subexpressions. For example, $\mathit{fv}(\text{IF}(e_1, e_2, e_3)) = \mathit{fv}(e_1) \cup \mathit{fv}(e_2) \cup \mathit{fv}(e_3)$. As another example, $\mathit{fv}(\text{APPLY}(f, e_1, \ldots, e_n)) = \mathit{fv}(e_1) \cup \cdots \mathit{fv}(e_n)$.

[2] And in addition, any variable that appears free in e also appears free in SET(x, e).

And we can also formalize the conjecture:

- Whenever $\langle e, \xi, \phi, \rho \rangle \Downarrow \langle v, \xi', \phi, \rho' \rangle$ and $y \in \text{dom } \xi$ and $y \notin \mathit{fv}(e)$, then $\xi'(y) = \xi(y)$.

This is the form of the conjecture that will be most useful for your solutions.

How to get started and what a solution should look like

If you're lucky, you'll have a flash of insight that will tell you exactly what is wrong with the conjecture—and you'll have your counterexample. But that is neither likely nor necessary to solve the problem. This problem should be solved by applying the method "How to attempt a metatheoretic proof" in the book in section~1.7.3, which starts on page~62. So, **to get started, attempt a case of a proof for each rule in Simplified Impcore.**

- Begin with the cases for rules FormalAssign and GlobalAssign, which you have to do anyway for part b. **Each case should follow the template in section~1.7.3,** with its six numbered steps.
- Keep going with other cases, one for each rule of Simplified Impcore, until you find a case for which the proof fails. You can use insight here, or you can just plod through until you find a case that fails.
- Once you've found a case for which the proof fails, **explain why**. That's part c. If you're not certain why it fails, find the counterexample, then go back to part c.
- Still working with the case for which the proof fails, you want to exhibit a counterexample. The counterexample refutes the conjecture. (That is, it demonstrates conclusively that the conjecture is no good.)

  To exhibit a counterexample you must exhibit a choice of e, v, and five environments for which the evaluation judgment holds—but in which evaluating e changes the value of a defined global variable, even though the variable does not appear free in e. The syntactic form of e is going to be an instance of the form in the conclusion of the rule that fails. For example, if the IfTrue case were to fail, then e would be an If expression. (*Hint: it doesn't and it's not.*)

  Once you find an expression e that refutes the conjecture, you need to say the environments in which e is to be evaluated. Environments ξ and ρ can be tiny, so they must be written out in full. But writing out ϕ in full would be tiresome, so instead, you may say "ϕ contains all the functions in the initial basis, plus the following additional function definitions," and then show the code for the definitions. (If you solve the problem without any additional definitions, that is OK.)

  The counterexample is part a.
- The last part of the problem, part d, is to show that you understand what is going on. Write out your explanation in a sentence or two.

Your solution should be written in two parts:

- The cases for the proof that you attempted. At minimum these must include the two successful `set` cases from part b, plus the failing case from part c. If you proved other cases successfully, you are welcome to include them.
- Answers to parts a, b, c, and d in an organized sequence.
  - The answer to part a must include the e, ξ, ϕ, ρ, v, ξ', and ρ' that refute the conjecture. A few words of explanation are also appropriate.

- The answer to part b should simply say that the two cases are included.
- The answer to part c should identify the case (rule) for which the proof fails, and should include a short explanation as to why the proof doesn't go through.
- The answer to part d should include a sentence or two suitable for a beginning programmer (e.g., CS 11 student).

Note: Free variables, formal parameters, and actual parameters

If e calls a function f, then f's variables (its *formal* parameters and any global variables it mentions), are *not* considered free in e. Only variables mentioned in the *actual* parameters appear free in e. If you are not confident on the distinction between formal and actual parameters, here is an explanation:

Formal and actual parameters

A function's **formal** parameters are the *names* of the parameters in its *definition*. For example, suppose I define function `square`:

```
(define square (x)
  (* x x))
```

A function's formal parameters never change; function `square` always has exactly one formal parameter, and it is named `x`.

Actual parameters are the *values* of the *arguments* given to a function at a *call site* (function application). These values, when the function is applied, will be the values that the formal parameters refer to. For example, if I write

```
(square 10)
```

then when this code is evaluated, `square` will be called with actual parameter 10. In the body of `square`, for this application, environment ρ will bind `x` to 10.

Actual parameters may differ at each call, even at a single call site. For example, in this code,

```
(while (> n 0)
  (begin
    (println (square n))
    (set n (- n 1))))
```

function `square` is called multiple times, and its *actual* parameter is different at each call.

Extra credit: Eliminating `begin`

Simplified Impcore has neither `while` nor `begin`. You already have an idea that you can often replace `while` with recursion. For extra credit, show that you can replace `begin` with function calls: complete exercise~14 on page~81 of *Build, Prove, and Compare*.

How to organize and submit your homework

Create files `awk-icon.imp` and `theory.pdf`. Please leave your name out of your PDF—that will enable your work to be graded anonymously.

You will turn in your code for exercise~16 (d) in file `awk-icon.imp`; everything else goes into `theory.pdf`. For the `theory.pdf`, you could consider using LaTeX, but unless you already have experience using LaTeX to typeset mathematics, it's a bad idea. We recommend that you write your theory homework by hand, then scan or photograph it, as described in the syllabus.

If you do already know LaTeX and you wish to use it, you may benefit by emulating our Latex source code for a simple proof system or Sam Guyer's LaTex source code for typesetting operational semantics. You might also like Matthew Ahrens's video tutorial on typesetting proof trees.

Please also create a file called `README`, in which you tell us anything else you think is useful for us to know. We provide a template for your `README`; it's online.

As soon as you have the files for all parts, `cd` into the appropriate directory and run `submit105-opsem` to submit a preliminary version of your work. You'll need files `README`, `awk-icon.imp`, and `theory.pdf`, but preliminary versions are good enough. Keep submitting and resubmitting until your work is complete; we grade only the last submission.

When you submit `theory.pdf`, the `provide` program should email you a copy of the PDF. Check the email and be sure that the PDF opens and displays what you expect. If there is a problem with the PDF, resubmit the file or ask for help on Piazza.

To help us read your work, we need for you to organize your answers carefully:

- The answer to each question must *start on a new page*.
- The theory answers *must appear in this order*: exercises 10, 11, R, 16 (parts a, b, and c), 12, 13 (part a), and finally F. If you choose to complete exercise~14 (the extra-credit problem shown above), put it last, after F.

How your work will be evaluated

Below is an extensive list of criteria for judging semantics, rules, derivations, and metatheoretic proofs. As always, you are aiming for the left-hand column, you might be willing to settle for the middle column, and you want to avoid the right-hand column.

Talking operational semantics (part A)

	Exemplary	Satisfactory	Must Improve
Talking	• Explanations are correct and use only words a beginning programmer would understand. • Formalism is completely correct; all primes, subscripts, and quantifiers are used correctly. • Claimed differences with Impcore are supported by example code (which may be just a fragment; complete, running code is not necessary). • When a rule is claimed to be effectively the same as the original, the solution explains why the notational differences are irrelevant.	• Explanations are correct, but they use words that a beginning programmer might not understand, like "environment." • Explanations are almost correct, and course staff can see what went wrong. • Formalism would be correct except for problems with quantifiers. (Quantifiers may be missing or may be in the wrong places, or $\forall$ may be confused with $\exists$.) • Formalism would be correct except for issues with subscripts or primes. • Claimed differences with Impcore are not supported by any code. • A rule is correct claimed to be effectively the same as the original, but the solution does not explain why the notational differences are irrelevant.	• There are explanations that the course staff can't easily relate to the formalism. • There are explanations that the course staff can't understand. • Formalism wouldn't be correct even if quantifiers, subscripts, and primes were corrected. • Differences with Impcore are not identified correctly.

Changed rules of Impcore (part B, exercise 16, parts (a) and (b))

	Exemplary	Satisfactory	Must Improve
Rules	• Every inference rule has a single conclusion which is a judgment form of the operational semantics. • In every inference rule, every premise is either a judgment form of the operational semantics or a simple mathematical predicate such as equality or set membership. • In every inference rule, if two states, two environments, or two of any other thing *must* be the same, then they are notated using a *single* metavariable that appears in multiple places. (Example: ρ or σ) • In every inference rule, if two states, two environments, or two of any other thing *may* be different, then they are notated using different metavariables. (Example: ρ and ρ') • New language designs use or change just enough rules to do the job. • Inference rules use one judgment form per syntactic category.	• In every inference rule, two states, two environments, or two of any other thing *must* be the same, yet they are notated using different metavariables. However, the inference rule includes a premise that these metavariables are equal. (Example: $\rho_1 = \rho_2$) • A new language design has a few too many new or changes a few too many existing rules. • Or, a new language design is missing a few rules that are needed, or it doesn't change a few existing rules that need to be changed.	• Notation that is presented as an inference rule has more than one judgment form or other predicate below the line. • Inference rules contain notation above the line that does not resemble a judgment form and is not a simple mathematical predicate. • Inference rules contain notation, either above or below the line, that resembles a judgment form but is not actually a judgment form. • In every inference rule, two states, two environments, or two of any other thing *must* be the same, yet they are notated using different metavariables, and nothing in the rule forces these metavariables to be equal. (Example: ρ and ρ' are both used, yet they must be identical.) • In some inference rule, two states, two environments, or two other things *may* be different, but they are notated using a single metavariable. (Example: using ρ everywhere, but in some places, ρ' is needed.) • In a new language design, the number of new or changed rules is a lot different from what is needed. • Inference rules contain a mix of judgment forms even when describing the semantics of a single syntactic category.

Exemplary	Satisfactory	Must Improve

Program to probe Impcore/Awk/Icon semantics (part B, exercise 16, part (d))

	Exemplary	Satisfactory	Must Improve
Semantics	• The program which is supposed to behave differently in Awk, Icon, and Impcore semantics behaves exactly as specified with each semantics.	• The program which is supposed to behave differently in Awk, Icon, and Impcore semantics behaves almost exactly as specified with each semantics.	• The program which is supposed to behave differently in Awk, Icon, and Impcore semantics gets one or more semantics wrong. • The program which is supposed to behave differently in Awk, Icon, and Impcore semantics looks like it is probably correct, but it does not meet the specification: either running the code does not print, or the last thing printed is not the 0 or 1 called for in the problem.

Derivations (part C, exercises 12 and 13)

	Exemplary	Satisfactory	Must Improve
Derivations	• In every derivation, every utterance is either a judgment form of the operational semantics or a simple mathematical predicate such as equality or set membership. • In every derivation, every judgement follows from instantiating a rule from the operational semantics. (Instantiating means substituting for meta variables.) The judgement appears below a horizontal line, and above *that* line is one derivation of each premise. • In every derivation, equal environments are notated equally. If both ρ and ρ' appear, they must *not* be known to be equal. • Every derivation takes the form of a tree. The root of the tree, which is written at the bottom, is the judgment that is derived (proved). • In every derivation, new bindings are added to an environment exactly as and when required by the semantics.	• In one or more derivations, there are a few horizontal lines that appear to be instances of inference rules, but the instantiations are not valid. (Example: rule requires two environments to be the same, but in the derivation they are different.) • In a derivation, the semantics requires new bindings to be added to some environments, and the derivation contains environments extended with the right new bindings, but not in exactly the right places.	• In one or more derivations, there are horizontal lines that the course staff is unable to relate to any inference rule. • In one or more derivations, there are many horizontal lines that appear to be instances of inference rules, but the instantiations are not valid. • Environments in intermediate or final states have primes or subscripts not found in the initial environment, and there is no unknown derivation (or unknown subexpression) whose result could account for a prime or a subscript. • A derivation is called for, but course staff cannot identify the tree structure of the judgments forming the derivation. • In a derivation, the semantics requires new bindings to be added to some environments, and the derivation contains environments extended with new bindings, but the new bindings in the derivation are not the bindings required by the semantics. (Example: the semantics calls for a binding of *answer* to 42, but instead *answer* is bound to 0.) • In a derivation, the semantics requires new bindings to be added to some environments, but the derivation does not contain any environments extended with new bindings.

Exemplary	Satisfactory	Must Improve

Metatheory (part C, exercise F)

	Exemplary	Satisfactory	Must Improve
Metatheory	• The counterexample includes everything mentioned in the evaluation judgment: expression, result, and all environments. • Proofs by induction explicitly identify the induction hypothesis. • Metatheoretic proofs operate by structural induction on derivations, and derivations are named. • Metatheoretic proofs classify derivations by case analysis over the final rule in each derivation. Cases with similar proofs are grouped together. • When a problem calls for a complete metatheoretic proof, the case analysis covers every possible derivation. • Failed cases for metatheoretic proof explain the failure even in the presence of a good induction hypothesis.	• The counterexample includes an expression, but it omits the result or one or more value environments. • A proof by induction is not explicit about what the induction hypothesis is, but course staff can figure it out. • Metatheoretic proofs operate by structural induction on derivations, but derivations and subderivations are not named, so course staff may not be certain of what's being claimed. • Metatheoretic proofs classify derivations by case analysis over the final rule in each derivation, but the grouping of the cases does not bring together cases with similar proofs. • Failed cases for metatheoretic proof are correctly identified, but they blame the induction hypothesis for the failure.	• The counterexample doesn't work. • In a proof by induction, course staff cannot figure out what the induction hypothesis is. • Metatheoretic proofs don't use structural induction on derivations (**serious fault**). • Metatheoretic proofs are missing many cases (**serious fault**). • Course staff cannot figure out how metatheoretic proof is broken down by cases (**serious fault**). • A problem calls for a complete metatheoretic proof, but the cases that are presented don't cover every possible derivation. • Failed cases for metatheoretic proof are not correctly identified.