Assignment: Operational Semantics

COMP 105

Due Tuesday, February 5, 2019 at 11:59PM

Contents

Part A: Reading comprehension (individual work, 10 percent) 1
Part B: Adding local variables to the interpreter (work with a partner, 23 percent) 3
Part C: Operational semantics, derivations, and metatheory (individual work, 67 percent) . . 5
Organizing the answers to Part C 7
Extra credit: Eliminating `begin` 7
How to organize and submit your work 8
How your work will be evaluated 8
    Adding local variables to Impcore (exercise 33) 8
    Operational semantics 9

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
- Connecting operational semantics with code in compilers or interpreters

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.

Part A: Reading comprehension (individual work, 10 percent)

Before starting the other problems below, answer these questions. You can download them[1].

For questions 1–7, please read pages 18–26 (the book sections on environments and on operational semantics of expressions). These questions are multiple-choice. Questions 1 to 3 offer one set of choices, and questions 4 to 6 offer another set of choices.

1. ξ is an environment that maps names to

   (a) *only* user-defined functions.
   (b) *only* the values of formal parameters.
   (c) *both* primitive and user-defined functions.
   (d) the values of *both* global variables and formal parameters.
   (e) *only* primitive functions.
   (f) *only* the values of global variables.

2. ϕ is an environment that maps names to

   (a) *only* user-defined functions.
   (b) *only* the values of formal parameters.
   (c) *both* primitive and user-defined functions.
   (d) the values of *both* global variables and formal parameters.
   (e) *only* primitive functions.
   (f) *only* the values of global variables.

3. ρ is an environment that maps names to

   (a) *only* user-defined functions.
   (b) *only* the values of formal parameters.
   (c) *both* primitive and user-defined functions.
   (d) the values of *both* global variables and formal parameters.
   (e) *only* primitive functions.
   (f) *only* the values of global variables.

4. In the operational semantics, what kind of a thing does the metavariable e stand for?

   (a) an environment
   (b) an Impcore variable
   (c) an elaboration
   (d) an expression
   (e) a value
   (f) a function

5. In the operational semantics, what kind of a thing does the metavariable v stand for?

   (a) an environment
   (b) an Impcore variable
   (c) an elaboration
   (d) an expression
   (e) a value
   (f) a function

[1] ./cqs.opsem.txt

6. In the operational semantics, what kind of a thing does the phrase $\rho\{x \mapsto 7\}(x)$ stand for?

   (a) an environment
   (b) an Impcore variable
   (c) an elaboration
   (d) an expression
   (e) a value
   (f) a function

7. In the operational semantics, what kind of a thing does the phrase $\rho\{x \mapsto 7\}\{x \mapsto 8\}$ stand for?

   (a) an environment
   (b) an Impcore variable
   (c) an elaboration
   (d) an expression
   (e) a value
   (f) a function

Questions 8 and 9 are also based on pages 18–26. Please answer a number.

8. How many rules have the IF syntactic form in the conclusion?

9. How many rules have the APPLY syntactic form in the conclusion? (Look at all the rules in the section, not just the summary on page 81.)

Now let's understand a subtle point about rules. Study the FormalVar and FormalAssign rules starting on page 21. In FormalVar, the initial and final states have the same ρ. But in the FormalAssign rule, there is an initial state with ρ, an intermediate state with ρ', and a final state with $\rho'\{x \mapsto v\}$. Answer these questions:

10. In FormalVar, the initial and final state have the same ρ because

    (a) Evaluating x might change the value of some formal parameter.
    (b) Evaluating x might not change the value of any formal parameter.
    (c) Evaluating x doesn't change the value of any formal parameter.

11. In FormalAssign, there is an intermediate state with ρ' (rho-prime) because

    (a) Evaluating e might change the value of some formal parameter.
    (b) Evaluating e might not change the value of any formal parameter.
    (c) Evaluating e doesn't change the value of any formal parameter.

When a rule is used in a derivation, it doesn't look exactly the way it looks in isolation. Regrettably, section 1.6.1, which starts on page 58 does not really explain how to construct a derivation. But look at the example derivation on page 59, and answer this question:

12. The same ρ is used throughout the derivation because

    (a) Every subexpression is known, and because there is no unknown subexpression, there is no need for a ρ' (rho-prime).
    (b) No part of the evaluation changes the value of a formal parameter.
    (c) The example derivation takes a shortcut and doesn't actually conform to the rules.

Part B: Adding local variables to the interpreter (work with a partner, 23 percent)

Related reading: section 1.5, particularly section 1.5.2, which starts on page 44. These pages walk you through the implementation of the operational semantics.

We use operational semantics in part because it's a superior way to write things we'd like to code. This exercise helps you understand how operational semantics is coded, and how language changes can be realized in C code. You will do exercise 33 from page 87 of the *Build, Prove, and Compare* book. *We recommend that you solve this problem with a partner, but this solution must be kept separate from your other solutions. Your programming partner, if any, must not see your other work.*

Here's an example of a function that uses a local variable:

```
; (sqrt n) approximates the square root of natural number n.
; It returns the largest integer whose square is no greater than n.

(define sqrt (n)
   [locals i]
   (begin
     (while (<= (* i i) n)
        (set i (+ i 1)))
     (- i 1)))
```

For information on pair programming, consult the syllabus[2], the reading[3], and some timeless advice for pair programmers[4].

- Get your copy of the code from the book by running

  ```
  git clone homework.cs.tufts.edu:/comp/105/build-prove-compare
  ```

  or if that doesn't work, from a lab or linux machine, try

  ```
  git clone /comp/105/build-prove-compare
  ```

  You can find the source code from Chapter 1 in subdirectory `bare/impcore` or `commented/impcore`. The `bare` version, which we recommend, contains just the C code from the book, with simple comments identifying page numbers. The `commented` version, which you may use if you like, includes part of the book text as commentary.

- We provide new versions of `all.h`, `definition-code.c`, `parse.c`, `printfuns.c`, and `tableparsing.c` that handle local variables. These versions are found in subdirectory `bare/impcore-with-locals`. There are not many changes; to see what is different, try running

  ```
  diff -r bare/impcore bare/impcore-with-locals
  ```

  You may wish to try the `-u` or `-y` options with `diff`. You may also wish to try `colordiff`.

  In the directory `bare/impcore-with-locals`, you can build an interpreter by typing `make`. The interpreter you build will parse definitions containing local variables, but it will ignore the local variables. To get local variables working, you'll make these changes:

[2] ../syllabus.html#how-do-pair-programming-interactions-work
[3] ../readings/pairs.pdf
[4] http://www.cs.tufts.edu/comp/40-2011f/readings/other-pair.html

  - In `eval.c`, you will modify the evaluator to give the right semantics to local variables. A local variable that has the same name as a formal parameter should hide that formal parameter, as in C.
  - You may also modify other files as you see fit.

  To build a list of values, you may wish to use function `mkVL` in file `list-code.c`.

- This exercise is **not** "coding a function from scratch." It is modifying a large, existing program. Therefore, *the nine-step design process that we usually recommend does not apply to this exercise.* We do not expect you to submit anything beyond the modified interpreter.
- Create a file called `README` in your `impcore-with-locals` directory. Describe your solution in the README.

Part C: Operational semantics, derivations, and metatheory (individual work, 67 percent)

Related reading:

- For an example of a derivation tree, see page 59.
- For rules of operational semantics, see section 1.4, which starts on page 18. The most important rules are summarized on pages 81–82.
- For metatheory, see section 1.6.2, which starts on page 59.

These exercises are intended to help you become fluent with operational semantics. *Do not share your solutions with any programming partners.* We encourage you to discuss ideas, but *no other student may see your rules, your derivations, or your code*. If you have difficulty, find a TA, who can help you work a couple of similar problems.

Do exercise 13 on page 82 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.

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.

Do exercise 14 on page 82 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.

Do exercises 21 and 22 on page 83 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 these exercises, 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 crazy.)

Exercise 22 does involve coding from scratch, and it could involve new functions. However, 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).

In exercise 22, we will assess your results by running your code 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.

Do exercise 20 on page 83 of *Build, Prove, and Compare*. In this exercise you prove that given a set of environments, the result of evaluating any expression e is completely determined. That is, in any given starting state, evaluation produces the same results every time. This proof requires you to raise your game again, 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.

Why do 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."

These proofs are stylized. To tackle the problem, assume you have two valid derivations with the same e and the same environments on the left, but different v's on the right—let's call them v_1 and v_2. You then prove that if both derivations are valid, $v_1 = v_2$. In other words, no matter what e is, you show that whenever $\langle e, \xi, \phi, \rho \rangle \Downarrow \langle v_1, \xi', \phi, \rho' \rangle$ and $\langle e, \xi, \phi, \rho \rangle \Downarrow \langle v_2, \xi'', \phi, \rho'' \rangle$, it is also true that $v_1 = v_2$.

The structure of your proof will resemble the structure given in section 1.6.3, which starts on page 61,

but because your proof involves reasoning about *two* derivations, it will be a little more complex. Your proof will proceed by induction on just one of the two derivations. You pick which one.

Note well that the theorem you are setting out to prove applies only to valid derivations that begin with the *same* initial state, including environments. But the conclusion of the theorem tells you only that the *values* are the same—it says nothing about the environments. If you set out naively, you'll find yourself in trouble on some of the induction steps. I know of two ways forward:

- Write a separate proof about the environments (not recommended).
- Find a *stronger* theorem that can also serve as an induction hypothesis, prove it, and then show that "Impcore is deterministic" follows as a corollary (recommended).

Whichever you choose, *be explicit about your induction hypothesis*.

To reduce bureaucracy, you will do this proof in **Theoretical Impcore**. Theoretical 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 Theoretical Impcore reduces the number of cases to a manageable number. This step will relieve some of the tedium (which, in this sort of proof, is regrettably common).

Organizing the answers to Part C

For these exercises you will turn in two files: `theory.pdf` and `awk-icon.imp`. For file `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[5].

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[6] or Sam Guyer's LaTex source code for typesetting operational semantics[7]. You might also like Matthew Ahrens's video tutorial on typesetting proof trees[8].

To help us read your answers to Part C, we need for you to organize them carefully:

- The answer to each question must *start on a new page*.
- The theory answers *must appear in this order*: exercises 13, 14, 21, and finally 20.
- Your answer to exercise 22 must be in file `awk-icon.imp`.

Extra credit: Eliminating `begin`

Theoretical 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.

Assume that ϕ binds the function `second` according to the following definition:

```
(define second (x y) y)
```

[5] http://www.cs.tufts.edu/comp/105/syllabus.html#then-how-should-theory-homework-be-written
[6] ../handouts/noset.tex
[7] ../handouts/latexexample.tex
[8] https://www.youtube.com/watch?v=zxVAi4L3y4Y&t=232s

I claim that if e_1 and e_2 are arbitrary expressions, you can always write `(second` e_1 e_2`)` instead of `(begin` e_1 e_2`)`. For extra credit, answer any or all of the following questions:

- **X1.** Using evaluation judgments, take the claim "you can always write `(second` e_1 e_2`)` instead of `(begin` e_1 e_2`)`" and restate the claim in precise, formal language.

  Hint: The claim is related to the claims in exercises 14 and 15 on page 82 in the Impcore chapter.

- **X2.** Using operational semantics, prove the claim.

- **X3.** Define a translation for `(begin` $e_1 \cdots e_n$`)` such that the translated code behaves exactly the same as the original code, but in the result of the translation, every remaining `begin` has exactly two subexpressions. For example, you might translate

  ```
  (begin e1 e2 e3)
  ```

  into

  ```
  (begin e1 (begin e2 e3))
  ```

  You may use any notation you like, but the cleanest way to define the translation is by using algebraic laws.

Once you've defined the translation in step X3, you'll be ready to write a translation that eliminates `begin` entirely. But that translation is more appropriate to next week's homework.

How to organize and submit your work

Before submitting code, **test what you can**. We do not provide any tests; you write your own. All code can be fully tested except the code for exercise 22.

- To complete part A, which you do by yourself, download the questions[9], then edit the answers into the file `cqs.opsem.txt`. If your editor is not good with Greek letters, you can spell out their names: ξ is "xi," ϕ is "phi," and ρ is "rho."

  You won't submit part A until you also have the files for part C.

- To submit part B, which you may have done with a partner, `cd` into `bare/impcore-with-locals`. The directory should contain your code and a `README` file that documents your solution.

  As soon as you have these files, run `submit105-opsem-pair` to submit a preliminary version of your work. Keep submitting until your work is complete; we grade only the last submission. *Only one partner should submit.*

- To complete part C, which you do by yourself, 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.

  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 at http://www.cs.tufts.edu/comp/105/homework/opsem-README-template.

  As soon as you have the files for parts A and C, `cd` into the appropriate directory and run `submit105-opsem-solo` to submit a preliminary version of your work. You'll need files `README`, `cqs.opsem.txt`, `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.

[9] ./cqs.opsem.txt

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.

How your work will be evaluated

Adding local variables to Impcore (exercise 33)

Everything in the general coding rubric[10] applies, but we will focus on three areas specific to this exercise:

	Exemplary	**Satisfactory**	**Must Improve**
Locals	• Change to interpreter appears motivated either by changing the semantics as little as possible or by changing the code as little as possible. • Local variables for Impcore pass simple tests.	• Course staff believe they can see motivation for changes to interpreter, but more changes were made than necessary. • Local variables for Impcore pass some tests.	• Course staff cannot understand what ideas were used to change the interpreter. • Local variables for Impcore pass few or no tests.
Naming	• Where the code implements math, the names of each variable in the code is either the same as what's in the math (e.g., `rho` for ρ), or is an English equivalent for what the code stands for (e.g., `parameters` or `parms` for ρ).	• Where the code implements math, the names don't help the course staff figure out how the code corresponds to the math.	• Where the code implements math, the course staff cannot figure out how the code corresponds to the math.
Structure	• The code is so clear that course staff can instantly tell whether it is correct or incorrect. • There's only as much code as is needed to do the job. • The code contains no redundant case analysis.	• Course staff have to work to tell whether the code is correct or incorrect. • There's somewhat more code than is needed to do the job. • The code contains a little redundant case analysis.	• From reading the code, course staff cannot tell whether it is correct or incorrect. • From reading the code, course staff cannot easily tell what it is doing. • There's about twice as much code as is needed to do the job. • A significant fraction of the case analyses in the code, maybe a third, are redundant.

[10] ../coding-rubric.html

Exemplary	Satisfactory	Must Improve

Operational semantics

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.

Changed rules of Impcore (exercise 21)

	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 (exercise 22)

	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 it prints two or more times.

Derivations (exercises 13 and 14)

	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 (exercise 20)

	Exemplary	**Satisfactory**	**Must Improve**
Metatheory	• 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. The case analysis includes every possible derivation, and cases with similar proofs are grouped together.	• 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. The case analysis includes every possible derivation, but the grouping of the cases does not bring together cases with similar proofs.	• Metatheoretic proofs don't use structural induction on derivations (**serious fault**). • Metatheoretic proofs have incomplete case analyses of derivations. • Metatheoretic proofs are missing many cases (**serious fault**). • Course staff cannot figure out how metatheoretic proof is broken down by cases (**serious fault**).