CS345H: Programming Languages

Lecture 8: Operational Semantics II

Thomas Dillig

Outline

- We will discuss semantics of remining (interesting) L expressions
- Will look at one more formalism for specifying meaning today

Back to Operational Semantics

- We are still missing semantics for key constructs in the L programming language
- Let's start with the if expression: `if e1 then e2 else e3`.
- Recall meaning: If `e1` evaluates to a non-zero integer, the meaning of the expression is `e2`, otherwise `e3`
- Any ideas on how to write this as an operational semantics rule?

Operational Semantics of Conditionals

- Difficulty: What happens depends on whether `e1` evaluates to 0 or not.
- Solution: Write two rules, one for the case where `e1` evaluates to 0 and one for the case whenre `e1` evaluates to a non-zero integer.
- What if `e1` evaluates to 0?

$$\frac{\begin{array}{c} E \vdash e_1 : 0 \\ E \vdash e_3 : e' \end{array}}{E \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : e'}$$

Operational Semantics of Conditionals Cont.

- What if `e1` evaluates to a non-zero integer?

$$\frac{\begin{array}{c} E \vdash e_1 : \text{non-zero integer} \\ E \vdash e_2 : e' \end{array}}{E \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : e'}$$

- Upshot: Can encode choice by giving multiple rules for same construct
- But need to make sure at most one rule can apply at any point for deterministic semantics
- Deterministic Semantics: Every program evaluates to at most one value

Operational Semantics of Function Definitions

- Recall: In L, function definitions of the form `fun f with x1,...,xn=e in...` are equivalent to `let f = lambda x1...lambda xn.e in ...`
- To define the meaning of a function definition, we can either repeat the lambda and let binding rules in one rule or rewrite the function definition into let and lambda's and invoke the existing rules
- We will do the latter:

$$\frac{E \vdash \text{ let } f = \text{ lambda } x_1 \ldots \text{ lambda } x_n.e_1 \text{ in } e_2 : e}{E \vdash \text{ fun } f \text{ with } x_1, \ldots, x_n = e_1 \text{ in } e_2 : e}$$

- This only works if there are no circular reductions!

Operational Semantics of Variable-Length Expressions

- The trick we just used to give meaning to function definitions is also useful for giving meaning to variable-length expressions.
- Consider the following grammar for a list of integers:

$$\begin{array}{lcl} S & \rightarrow & [E] \\ E & \rightarrow & \text{int } E \mid \text{int} \end{array}$$

- Example strings in L(S): `[3], [2 3 4], [1 3],...`
- Suppose we want to define the meaning of a list of integers as their sum: How can we write operational semantics for this mini-language?

Operational Semantics of Variable-Length Expressions

- Observation: Difficulty caused by unknown length of list
- Let's write operational semantics for a list of length 2:

$$\frac{}{\vdash [i_1 \ i_2] : i_1 + i_2}$$

- Solution: Think recursively! The sum of a list of k integers can be obtained by removing the first integer, computing the sum of the remainder and adding the two values
- This translates into two rules: Base case and inductive case

Operational Semantics of Variable-Length Expressions

- Base case: List with one integer

$$\frac{}{\vdash [i] : i}$$

- Inductive Case: List with at least two integers

$$\frac{\vdash [R] : i_2}{\vdash [i_1, R] : i_1 + i_2}$$

- Upshot: To give semantics to variable-length expression, decompose recursively into inductive case(s) and base case(s)
- Observe that it is possible to encode computation in this formalism, we will (briefly) see this again towards the end of the class

Alternative Semantics

- We can also define the meaning of a list program as follows:
  Base case:

$$\frac{}{\vdash i : i}$$

  Inductive case:

$$\frac{\vdash e_1 : i_1 \quad \vdash : e_2 : i_2}{\vdash e_1 + e_2 : i_1 + i_2}$$

  Removing the brackets:

$$\frac{\vdash e : i}{\vdash [e] : i}$$

- Are these two semantics equivalent?

Operational Semantics of Application in L

- Last time we only gave operational semantics for the application base case: Two expressions:

$$\frac{\begin{array}{l} E \vdash e_1 : lambda \ x. \ e_1' \\ E \vdash e_1'[e_2/x] : e \end{array}}{E \vdash (e_1 \ e_2) : e}$$

- But the application can have any number of expressions in L. Example: `(x y z)` is a valid L expression with meaning `((x y) z)`
- Solution: Write inductive case for more than two expressions!

$$\frac{\begin{array}{l} E \vdash e_1 : \texttt{lambda} \ x.e_1' \\ E \vdash e_1'[e_2/x] : e \\ E \vdash (e \ R) : e' \end{array}}{E \vdash (e_1 \ e_2 \ R) : e'}$$

Operational Semantics of Application in L

- What about an application with one expression, such as `(x)`?
- This is not an application
- Observe: L syntax allows this to indicate associativity and precedence
- Question: What is the meaning (operational semantics rule) for `(x)`?
- Answer:

$$\frac{E \vdash e : e'}{E \vdash (e) : e'}$$

List Operations

- Let's also take a brief look at semantics for some list operations:
- Consider `!e`, which evaluated to the head of the list if `e` is a list and to `e` otherwise
- e is a list:

$$\frac{E \vdash e : [e_1, e_2]}{E \vdash !e : e_1}$$

- e is not a list:

$$\frac{E \vdash e : e_1 \ (e_1 \text{ not a list})}{E \vdash !e : e_1}$$

List Operations

- What about `e1@e2`, which evaluated to the list `[e1, e2]`?

$$\frac{\begin{array}{c} E \vdash e_1 : e_1' \\ E \vdash e_2 : e_2' (e_2' \text{ not Nil}) \end{array}}{E \vdash e_1 @ e_2 : [e_1', e_2']}$$

- e2 evaluates to Nil:

$$\frac{\begin{array}{c} E \vdash e_1 : e_1' \\ E \vdash e_2 : Nil \end{array}}{E \vdash e_1 @ e_2 : e_1'}$$

Congratulations!

- You can now understand every page in the L reference manual.
- For PA3, you will need to refer to the operational semantics of L in the manual to implement your interpreter.
- The manual is the official source for the semantics of L, not the reference interpreter!

Operational Semantics

- The rules we have written are known as large-step operational semantics
- They are called large step because each rule completely evaluates an expression, taking as many steps as necessary.
- Example: The plus rule

$$\frac{\begin{array}{c} E \vdash e_1 : i_1 \text{ (integer)} \\ E \vdash e_2 : i_2 \text{ (integer)} \end{array}}{E \vdash e_1 + e_2 : i_1 + i_2}$$

- Here, we evaluate both e_1 and e_2 to compute the final value in one (big) step
- Alternate formalism for giving semantics: small-step operational semantics

Small Step Operational Semantics

- Small-step operational semantics perform only one step of computation per rule invocation
- You can think of SSOS as "decomposing" all operations that happen in one rule in LSOS into individual steps
- This means: Each rule in SSOS has at most one precondition

Small-step Operational Semantics

- SSOS are easiest understood by an example. Consider the integer plus in L written in SSOS:
- Rule 1: Adding two integers

$$\frac{}{\langle c_1 + c_2, E\rangle \rightarrow \langle c_1 + c_2, E\rangle}$$

- Rule 2: Reducing first expression to an integer

$$\frac{\langle e_1, E\rangle \rightarrow \langle c, E'\rangle}{\langle e_1 + e_2, E\rangle \rightarrow \langle c + e_2, E'\rangle}$$

- Rule 3: Reducing second expression to an integer

$$\frac{\langle e, E\rangle \rightarrow \langle c_2, E'\rangle}{\langle c_1 + e, E\rangle \rightarrow \langle c_1 + c_2, E'\rangle}$$

SSOS in Action

- Let's use these rules to prove what the value of $(2+4)+6$ is:
- $\langle (2+4)+6, \_\rangle \rightarrow \langle 6+6, \_\rangle \rightarrow \langle 12, \_\rangle$

SSOS

- You can tell small-step operational semantics by the $\langle\rangle \rightarrow$ notation
- In contrast, LSOS have the $\vdash$: notation (at least in this class)
- SSOS are really (conditional) rewrite rules
- The β reduction of λ-calculus is a small-step semantics rule

SSOS of the Application

- Recall the large-step operational semantics:

$$\frac{\begin{array}{c} E \vdash e_1 : lambda\ x.\ e_1' \\ E \vdash e_1'[e_2/x] : e \end{array}}{E \vdash (e_1\ e_2) : e}$$

- What are equivalent SSOS?

$$\frac{\langle e_1'[e_2/x], E\rangle \rightarrow \langle e_3, E'\rangle}{\langle(\text{lambda } x.e_1'\ e_2), E\rangle \rightarrow \langle e_3, E'\rangle}$$

SSOS of the Application

- Recall the large-step operational semantics, evaluating e_1 made a difference:

$$\frac{\begin{array}{c} E \vdash e_1 : lambda\ x.\ e_1' \\ E \vdash e_1'[e_2/x] : e \end{array}}{E \vdash (e_1\ e_2) : e}$$

- What about in SSOS?
- For SSOS, other rules will rewrite the expression until it matches the form $lambda\ x.\ e_1'$

SSOS of let

- First try:

$$\frac{\langle e_2, E[x \leftarrow e_1]\rangle \rightarrow \langle e_3, \_\rangle}{\langle let\ x = e_1\ in\ e_2, E\rangle \rightarrow \langle e_3, E\rangle}$$

- But we want eager semantics: We want to evaluate e_1 before adding to the environment.
- We want a rule that evaluates e_1 as much as possible and only then applies the let rule:
- Notation: We will write $\widehat{e}$ to indicate that expression e has been evaluated as much as possible.

SSOS of let cont.

- Here are the two rules for eager let in SSOS:

$$\frac{\langle e_2, E[x \leftarrow \widehat{e_1}]\rangle \rightarrow \langle e_2, \_\rangle}{\langle let\ x = \widehat{e_1}\ in\ e_2, E\rangle \rightarrow \langle e_3, E\rangle}$$

$$\frac{\langle e_1, E\rangle \rightarrow \langle \widehat{e_1}, E'\rangle}{\langle let\ x = e_1\ in\ e_2, E\rangle \rightarrow \langle let\ x = \widehat{e_1}\ in\ e_2, E'\rangle}$$

Small-step vs. Big-step Semantics

- In big-step semantics, any rule may invoke any number of other rules in the hypothesis
- This means any derivation is a tree.
- In small-step semantics, each rule only performs one step of computation
- This means any derivation is a line

Advantages of SSOS

- The main advantage of SSOS is that it allows us to distinguish between non-terminating computation and undefined computation
- Recall: In BSOS, encountering an undefined expression, such as `3+"duck"` got us "stuck", i.e., we could never satisfy the hypothesis to reach a conclusion
- In SSOS, undefined expressions also get stuck,i.e. no rule applies

Advantages of SSOS Cont.

- But, consider the following program: `fun f with x = (f x) in (f 1)`.
  - In BSOS, we will "get stuck", i.e. we will never satisfy all hypothesis of the function invocation
  - In SSOS, we will have an infinite derivation line
- Upshot: SSOS allow us to distinguish non-termination from errors

Big vs. Small-Step Semantics

- The other big difference is that we can quantify the cost of a computation with the number of steps in a small-step derivation
- This allows us to talk about (some) notions of complexity when analyzing small-step semantics
- Main disadvantage of small step semantics is that they are less intuitive and and usually harder to write
- SSOS also always force one order, even if we would like to leave an order undefined

Conclusion

- We have seen two formalisms for specifying meaning of programs
- There are at least two more in common use: Denotational Semantics and Axiomatic Semantics
- However, operational semantics seem to be winning the "semantics wars"
- Why: Easier to understand and easier to prove (most) properties with them