Outline

- We will discuss semantics of remaining (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 where e1 evaluates to a non-zero integer.
- What if e1 evaluates to 0?
  \[ E \vdash e_1 : 0 \]
  \[ E \vdash e_3 : e' \]
  \[ E \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : e' \]

Operational Semantics of Conditionals Cont.

- What if e1 evaluates to a non-zero integer?
  \[ E \vdash e_1 : \text{non-zero integer} \]
  \[ E \vdash e_2 : e' \]
  \[ E \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : e' \]

- 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
  \[ \text{fun } f \text{ with } x_1, \ldots, x_n = e \ldots \] are equivalent to
  \[ \text{let } f = \text{lambda } x_1 \ldots \text{lambda } x_n. e \ldots \]
- 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:
  \[ E \vdash \text{let } f = \text{lambda } x_1, \ldots, x_n. e_1 \text{ in } e_2 : e \]
  \[ E \vdash \text{fun } f \text{ with } x_1, \ldots, x_n = e_1 \text{ in } e_2 : e \]
- 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:
  \[ S \rightarrow [E] \]
  \[ E \rightarrow \text{int } E \mid \text{int} \]
- 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 Application in L

- Last time we only gave operational semantics for the application base case: Two expressions:
  \[ E \vdash e_1 : e'_1 \]
  \[ E \vdash e'_1[e_2/x] : e \]
  \[ E \vdash (e_1 e_2) : e' \]
- 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!
  \[ E \vdash e_1 : \lambda x. e'_1 \]
  \[ E \vdash e'_1[e_2/x] : e \]
  \[ E \vdash (e R) : e' \]
  \[ E \vdash (e_1 e_2 R) : e' \]

Alternative Semantics

- We can also define the meaning of a list program as follows:
  Base case:
  \[ \vdash i : i \]
  Inductive case:
  \[ \vdash [R] : i_2 \]
  \[ \vdash [i_1, R] : i_1 + i_2 \]
- Observe that it is possible to encode computation in this formalism, we will (briefly) see this again towards the end of the class.

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:
    \[ E \vdash e : e' \]
    \[ E \vdash (e) : e' \]
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!

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.

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
  \[
  E \vdash e_1 : i_1 \text{ (integer)}
  \] 
  \[
  E \vdash e_2 : i_2 \text{ (integer)}
  \] 
  \[
  E \vdash e_1 + e_2 : i_1 + i_2
  \]
- 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.

List Operations

- Let’s also take a brief look at semantics for some list operations:
- Consider \(e_2\), which evaluated to the list \([e_1, e_2]\)?
  \[
  E \vdash e_1 : e_1'
  \] 
  \[
  E \vdash e_2 : e_2' (e_2' \text{ not Nil})
  \]
  \[
  E \vdash e_1@e_2 : [e_1', e_2']
  \]
- \(e_2\) evaluates to Nil:
  \[
  E \vdash e_1 : e_1'
  \] 
  \[
  E \vdash e_2 : \text{ Nil}
  \]
  \[
  E \vdash e_1@e_2 : e_1'
  \]

SSOS are easiest understood by an example. Consider the integer plus in L written in SSOS:

\[
E \vdash e_1 : e_1'
\]
\[
E \vdash e_2 : e_2'
\]
\[
E \vdash e_1 + e_2 : e_1' + e_2'
\]

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
  \[
  (e_1 + e_2, E) \rightarrow (e_1 + e_2, E)
  \]
- Rule 2: Reducing first expression to an integer
  \[
  (e_1, E) \rightarrow (e, E')
  \]
  \[
  (e_1 + e_2, E) \rightarrow (e + e_2, E')
  \]
- Rule 3: Reducing second expression to an integer
  \[
  (e, E) \rightarrow (e, E')
  \]
  \[
  (e_1 + e, E) \rightarrow (e_1 + e_2, E')
  \]
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 \(\beta\) reduction of \(\lambda\)-calculus is a small-step semantics rule

SSOS of the Application

- Recall the large-step operational semantics:
  \[
  E \vdash e_1 : \text{lambda } x. e'_1 \\
  E \vdash e'_1[x/e_2] : e \\
  E \vdash (e_1 e_2) : e
  \]
- What are equivalent SSOS?
  \[
  (\langle e'_1 e_2/x, E \rangle, E) \rightarrow (\langle e_3, E' \rangle)
  \]

SSOS of let

- First try:
  \[
  \langle e_2, E[x \leftarrow e_1] \rangle \rightarrow (e_3, E)
  \]
- 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 \(\hat{e}\) 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:
  \[
  \langle e_2, E[x \leftarrow e_1] \rangle \rightarrow (e_3, E)
  \]
  \[
  \langle e_1, E \rangle \rightarrow (\hat{e}_1, E')
  \]
  \[
  \langle \text{let } x = e_1 \text{ in } e_2, E \rangle \rightarrow \langle \text{let } x = \hat{e}_1 \text{ in } e_2, E' \rangle
  \]
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 \( 3 + \text{"duck"} \) 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: \( \text{fun } f \text{ with } x = (f \ x) \text{ in } (f \ 1) \).
  - 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.