CS345H: Programming Languages

Lecture 7: Operational Semantics I

Thomas Dillig

Outline

- Next Topic: Semantics
- How to specify meaning of syntax
- Will look at one formalism for this today

What does a program mean?

- We have learned how to specify syntax.
- Example: `let x = lambda lambda` is not a valid L program
- But we have not yet talked about what the meaning of a program is.
- First Question: What is the meaning of a program in L?
- Answer: The value the program evaluates to
- Example: `let x = 3 in x` Value: 3

How to specify meaning of programs

- Option 1: Don't worry too much
- Developer of language has some informal concept of the intended meaning, implement a compiler/interpreter that does whatever the language designers believe to be reasonable.
- Then, declare the meaning to be whatever the compiler produces
- A terrible idea

How to specify meaning of programs

- Why is this such a bad idea?
  - This approach promotes bugs/inconsistencies to expected behavior.
  - Hides specification of language in many implementation details
  - Makes it almost impossible to implement another compiler that accepts the same language
- Unfortunately, this is (still) a very common approach
- Languages designed this way: C, C++ (to some extent), Perl, PHP, JavaScript, ...

How to specify meaning of programs

- Option 2: Try to write out precisely the meaning of each language construct in documentation, then follow this description in implementation
- Example: Describe the meaning of `!e` in the L language:
- First attempt: "This evaluates to the head of e"
- What if e is not a list?
- Second attempt: "This evaluates to the head of e if e is a list, and to e otherwise"
- What if e is Nil? ...

How to specify meaning of programs: Option 2

- Written language is, by nature, ambiguous. It is very difficult to fully specify the meaning of all language constructs this way
- Easy to miss cases
- Results in long, complicated and difficult to understand specifications, but an improvement over no specification

Written specification in practice

- Let's look at the ISO C++ standard: 879 pages, page 116:

5.16 Conditional operator **[expr.cond]**

conditional-expression:
logical-or-expression
logical-or-expression ? *expression* : *assignment-expression*

1 Conditional expressions group right-to-left. The first expression is implicitly converted to `bool` (clause 4). It is evaluated and if it is `true`, the result of the conditional expression is the value of the second expression, otherwise that of the third expression. All side effects of the first expression except for destruction of temporaries (12.2) happen before the second or third expression is evaluated. Only one of the second and third expressions is evaluated.

2 If either the second or the third operand has type (possibly cv-qualified) `void`, then the lvalue-to-rvalue (4.1), array-to-pointer (4.2), and function-to-pointer (4.3) standard conversions are performed on the second and third operands, and one of the following shall hold:

— The second or the third operand (but not both) is a *throw-expression* (15.1); the result is of the type of the other and is an rvalue.

— Both the second and the third operands have type `void` the result is of type `void` and is an rvalue. [*Note:* this includes the case where both operands are *throw-expressions*. — *end note*]

3 Otherwise, if the second and third operand have different types, and either has (possibly cv-qualified) class type, an attempt is made to convert each of those operands to the type of the other. The process for determining whether an operand expression `E1` of type `T1` can be converted to match an operand expression `E2` of type `T2` is defined as follows:

— If `E2` is an lvalue: `E1` can be converted to match `E2` if `E1` can be implicitly converted (clause 4) to the type "reference to `T2`", subject to the constraint that in the conversion the reference must bind directly (8.5.3) to `E1`.

— If `E2` is an rvalue, or if the conversion above cannot be done:

— if `E1` and `E2` have class type, and the underlying class types are the same or one is a base class of the other: `E1` can be converted to match `E2` if the class of `T2` is the same type as, or a base class of, the class of `T1`, and the cv-qualification of `T2` is the same cv-qualification as, or a greater cv-qualification than, the cv-qualification of `T1`. If the conversion is applied, `E1` is changed to an rvalue of type `T2` ~~that still refers to the original source class object (or the appropriate subobject thereof). [Note: that is, no copy is made. — end note]~~ by copy-initializing a temporary of type `T2` from `E1` and using that temporary as the converted operand.

— Otherwise (i.e., if `E1` or `E2` has a nonclass type, or if they both have class types but the underlying classes are not either the same or one a base class of the other): `E1` can be converted to match `E2` if `E1` can be implicitly

Precisely Specifying Meaning

- Recall λ-calculus:
- To specify the meaning of expressions, we defined one single operation: β reduction
- Specifically, we wrote $\lambda x.e_1\ e_2 \to^{\beta} e_1[e_2/x]$
- Can read this as follows: If you see an expression of the form $\lambda x.e_1\ e_2$, you can compute its result as $e_1[e_2/x]$.

Operational Semantics

- Let's try the same in the language of arithmetic expression with the grammar:

$$S \to c \mid S_1 + S_2 \mid S_1 * S_2$$

- What is the meaning of an integer constant? The value of this integer
- More precisely: If we see an expression of the form c, its value is c
- We will write:

$$\vdash \mathtt{c} : c$$

- Read as: "we can prove for any expression of the form c
- that the meaning of this expression is c"

Operational Semantics Cont.

- How about the expression $S_1 + S_2$?
- $\vdash S_1 + S_2 :?$
- Problem: To describe the meaning of $S_1 + S_2$, we need to know the meaning (value) of S_1 and S_2
- Solution: Use hypotheses: We want to say "Assuming S_1 evaluates to c_1 and S_2 evaluates to c_2, the value of $S_1 + S_2$ is $c_1 + c_2$"
- We write this as:

$$\frac{\begin{array}{c}\vdash S_1 : c_1 \\ \vdash S_2 : c_2\end{array}}{\vdash S_1 + S_2 : c_1 + c_2}$$

Inference Rules

- This notation is known as inference rule:

$$\frac{\begin{array}{c}\text{Hypothesis 1} \\ \dots \\ \text{Hypothesis N}\end{array}}{\vdash \text{Conclusion}}$$

- This means "given hypothesis 1, ... N, the conclusion is provable"
- Example:

$$\frac{\begin{array}{c}\text{Miterm 1 grade} >= 70 \\ \dots \\ \text{Final grade} >= 140\end{array}}{\vdash \text{Final grade: A}}$$

Inference Rules cont.

- A hypothesis in an inference rule may use other rules
- Example:

$$\frac{\vdash S_1 : c_1 \qquad \vdash S_2 : c_2}{\vdash S_1 + S_2 : c_1 + c_2}$$

- You can tell this by a $\vdash$ in at least one of the hypotheses.
- Such rules are called inductive since they define the meaning of an expression in terms of the meaning of subexpressions.
- Rules that to not have $\vdash$ in any hypothesis are base cases
- A system with only inductive rules is nonsensical

Operational Semantics

- Back to the rule for $+$:

$$\frac{\vdash S_1 : c_1 \qquad \vdash S_2 : c_2}{\vdash S_1 + S_2 : c_1 + c_2}$$

- Let's focus on the first hypothesis $\vdash S_1 : c_1$.
- Question: Can you write $S_1 = c_1$?
- Answer: Yes, but now your first hypothesis is: " Assuming S_1 is the integer constant c_1" $\Rightarrow$ this rule no longer applies if, for example, $S_1 = 2 * 3$.
- Read $\vdash$ as "is provable by using our set of inference rules".

Operational Semantics and Order

- Important Point: This notation does not specify an order between hypothesis.
- This means that

$$\frac{\vdash S_1 : c_1 \qquad \vdash S_2 : c_2}{\vdash S_1 + S_2 : c_1 + c_2}$$

and

$$\frac{\vdash S_2 : c_2 \qquad \vdash S_1 : c_1}{\vdash S_1 + S_2 : c_1 + c_2}$$

have exactly the same meaning

Full Operational Semantics

- Here are the full operational semantics of the language

$$S \rightarrow c \mid S_1 + S_2 \mid S_1 * S_2$$

$$\vdash c : c$$

$$\frac{\vdash S_1 : c_1 \qquad \vdash S_2 : c_2}{\vdash S_1 + S_2 : c_1 + c_2}$$

$$\frac{\vdash S_1 : c_1 \qquad \vdash S_2 : c_2}{\vdash S_1 * S_2 : c_1 * c_2}$$

Using Operational Semantics

- Consider the expression $(21 * 2) + 6$
- Here is how to derive the value of this expression with the operational semantics:

$$\frac{\dfrac{\vdash 21 : 21 \qquad \vdash 2 : 2}{\vdash 21 * 2 : 42} \qquad \vdash 6 : 6}{\vdash (21 * 2) + 6 : 48}$$

- This is a formal proof that the expression $(21 * 2) + 6$ evaluates to 48 under the defined operational semantics
- Observe that these proofs have a tree structure: Each subexpression forms a new branch in the tree

Operational Semantics of L

- Let's try to give operational semantics to the L language:
- Start with integers: $\dfrac{\text{Integer } \mathtt{i}}{\vdash \mathtt{i} : i}$
- The `i` in the hypothesis and to the left of the colon is the syntactic number in the source code of L
- The i after the colon is the value of the integer i.
- This sounds nitpicky, but is important to understand this notation.

Operational Semantics of L

- Consider the (integer) plus expression in L:

$$\frac{\vdash e_1 : i_1 \text{ (integer)} \quad \vdash e_2 : i_2 \text{ (integer)}}{\vdash e_1 + e_2 : i_1 + i_2}$$

- Side remark: The hypothesis can be written in separate lines (but not when giving a derivation tree)
- Here, the hypotheses require that e_1 and e_2 evaluate to integers.
- Question: What happens if e_1 evaluates to a list?
- Answer: No rule applies and computation is "stuck". This means the L program does not evaluate to anything.
- In practice: This is a run-time error

Operational Semantics of L

- Integer minus:

$$\frac{\vdash e_1 : i_1 \text{ (integer)} \quad \vdash e_2 : i_2 \text{ (integer)}}{\vdash e_1 - e_2 : i_1 - i_2}$$

- Integer times:

$$\frac{\vdash e_1 : i_1 \text{ (integer)} \quad \vdash e_2 : i_2 \text{ (integer)}}{\vdash e_1 * e_2 : i_1 * i_2}$$

Operational Semantics of L

- On to the key construct: λ
- Let's write semantics for the simple application (e1 e2)
- Recall that this is only defined if e1 is a lambda expression.
- Hypothesis: $\vdash e_1 : lambda\ x.\ e_1'$
- Now, how do we evaluate (e1 e2) ? $\vdash e_1'[e_2/x] : e$
- Conclusion: $\vdash (e_1\ e_2) : e$
- Final rule:

$$\frac{\vdash e_1 : lambda\ x.\ e_1' \quad \vdash e_1'[e_2/x] : e}{\vdash (e_1\ e_2) : e}$$

Order of Evaluation

- What would change if we write:

$$\frac{\vdash e_1'[e_2/x] : e \quad \vdash e_1 : lambda\ x.\ e_1'}{\vdash (e_1\ e_2) : e}$$

- Answer: Nothing. The written order of hypotheses is irrelevant
- Observe: This rule does specify an order between hypothesis: $\vdash e_1 : lambda\ x.\ e_1'$ must be evaluated before $\vdash e_1'[e_2/x] : e$.
- This is the case because $\vdash e_1'[e_2/x] : e$ uses e_1' defined by hypothesis $\vdash e_1 : lambda\ x.\ e_1'$
- Important Point: Operational semantics can encode order, but not through syntactic ordering

The Lambda Rule

- Question: What would change if we write the hypothesis as

$$\frac{e_1 = lambda\ x.\ e_1' \quad \vdash e_1'[e_2/x] : e}{\vdash (e_1\ e_2) : e}$$

- Answer: This would still give semantics to `(lambda x.x 3)`, but no longer to `let y=lambda x.x in (y 3)`

The Lambda Rule cont.

$$\frac{\vdash e_1 : lambda\ x.\ e_1' \quad \vdash e_1'[e_2/x] : e}{\vdash (e_1\ e_2) : e}$$

- Observe that in this rule, we are not evaluating e_2 before substitution.
- Consider the following modified rule:

$$\frac{\vdash e_1 : lambda\ x.\ e_1' \quad \vdash e_2 : e_2' \quad \vdash e_1'[e_2'/x] : e}{\vdash (e_1\ e_2) : e}$$

- This also is a well-formed rule, but it gives a different meaning to the lambda expression

The Lambda Rule cont.

- Consider both rules:

$$\frac{\vdash e_1 : lambda\ x.\ e_1' \quad \vdash e_1'[e_2/x] : e}{\vdash (e_1\ e_2) : e}$$

$$\frac{\vdash e_1 : lambda\ x.\ e_1' \quad \vdash e_2 : e_2' \quad \vdash e_1'[e_2'/x] : e}{\vdash (e_1\ e_2) : e}$$

- Consider the expression `(lambda x.3 4+"duck")`:
  - Rule 1 evaluates this expression to "3"
  - Rule 2 "gets stuck" and returns no value since adding an integer and string is undefined (we have not given a rule)
- Two reasonable ways of defining application, but different semantics!

Call-by-name vs. Call-by-value

- Not evaluating the argument before substitution is known as call-by name, evaluating the argument before substitution as call-by-value.
- Languages with call-by-name: classic lambda calculus, ALGOL 60, L
- Languages with call-by-value: C, C++, Java, Python, FORTRAN, ...
- Advantage of call-by-name: If argument is not used, it will not be evaluated
- Disadvantage: If argument is uses k times, it will be evaluated k times!

Call-by-name vs. Call-by-value

- Consider the following expression in L syntax:
  `(lambda x.x+x+x (77*3-2))`
- Under call-by-name semantics, we substitute `(77*3-2)` for `x` and reduce the problem of evaluating `(lambda x.x+x+x (77*3-2))` to evaluating `((77*3-2)+(77*3-2)+(77*3-2))`
- We compute the value of `x` three times
- Under call-by-value semantics, we first evaluate `(77*3-2)` to 229 and then evaluate 229+229+229

Semantics of the let-binding

- Let's try to define the semantics of the let-binding in L:
  `let x = e1 in e2`
- One possibility:

$$\frac{\vdash e_1 : e_1' \qquad \vdash e_2[e_1'/x] : e}{\vdash \text{let } x = e_1 \text{ in } e_2 : e}$$

- What about the following definition?

$$\frac{\vdash e_2[e_1/x] : e}{\vdash \text{let } x = e_1 \text{ in } e_2 : e}$$

- Are these definitions equivalent?

Eager vs. Lazy Evaluation

- Evaluating e_1 before we know that it is used is called eager evaluation
- Waiting until we need it is lazy evaluation.
- These are analogous to call-by-name/call-by value in trade offs.

Definition of let bindings

- But currently there is one problem common to both the eager and lazy definition of the let binding.
- Consider the following valid L program:

```
let f =
  lambda x.  if x <= 0 then 1 else x*(f (x-1))
in (f 2)
```

- What happens if we use our definition of let on this expression? For brevity, let's use the lazy one here, but the same problem exists with the eager one:

$$\frac{\vdash (\mathtt{f}\ 2)[(\mathtt{lambda}\ \mathtt{x}.\mathtt{if}\ \mathtt{x} <= 0\ \mathtt{then}\ 1\ \mathtt{else}\ \mathtt{x} * (\mathtt{f}(\mathtt{x}-1))/\mathtt{f}] : ?}{\vdash \mathtt{let}\ \mathtt{f} = \mathtt{lambda}\ \mathtt{x}.\mathtt{if}\ \mathtt{x} <= 0\ \mathtt{then}\ 1\ \mathtt{else}\ \mathtt{x} * (\mathtt{f}(\mathtt{x}-1))\ \mathtt{in}\ (\mathtt{f}\ 2) : ?}$$

Let Binding

- We have already seen this problem when studying lambda calculus.
- But this time, we want to solve it. After all, who wants to use the Y-combinator for every recursive function!
- Solution: Add an environment to our rules that tracks mappings between identifiers and values
- Specifically, write the let rule as follows:

$$\frac{\begin{array}{c} E \vdash e_1 : e_1' \\ E[x \leftarrow e_1'] \vdash e_2 : e \end{array}}{E \vdash \text{let } x = e_1 \text{ in } e_2 : e}$$

Environments

- You can think of the environment as storing information to be used by other rules
- An environment maps keys to values
- Notation: $E[x \leftarrow y]$ means new environment with all mappings in E and the mapping $x \mapsto y$ added.
- If x was already mapped in E, the mapping is replaced
- Notation: $E(x) = y$ means bind value of key x in E to y. If no mapping $x \mapsto y$ exits in E, this "gets stuck"

Environments

- An environment adds extra information!
- In this rule:

$$\frac{\begin{array}{c} E \vdash e_1 : e_1' \\ E[x \leftarrow e_1'] \vdash e_2 : e \end{array}}{E \vdash \text{let } x = e_1 \text{ in } e_2 : e}$$

- Read the hypothesis $E \vdash e_1 : e_1'$ as: "Given environment E and expression e_1 and that it is provable that e_2 evaluates to e"
- Read the conclusion as: "Given environment E and expression let $x = e_1$ in e_2, this expression evaluates to e.

Environments

- Since we are no longer replacing the let-bound identifiers, we also need a base case for identifiers
- This will now use the environment:

$$\frac{\begin{array}{c} \text{Identifier id} \\ E(\text{id}) = e \end{array}}{E \vdash \text{id} : e}$$

- Adding the environment allows us now to be able to give (intuitive) meaning to recursive programs.

Environments Example

- Consider the L program `let x = 3 in x`
- Here is the proof that this program evaluates to 3:

$$\frac{E \vdash 3 : 3 \qquad \dfrac{\begin{array}{c} \text{Identifier x} \\ E[x \leftarrow 3](\mathsf{x}) = 3 \end{array}}{E[x \leftarrow 3] \vdash \mathsf{x} : 3}}{E \vdash \text{let } x = 3 \text{ in } x : 3}$$

Conclusion

- We have seen how to formally give meaning to programs
- The formalism we have studied is called large-step operational semantics
- Next time: Semantics for more L constructs and another alternative formalism for specifying meaning of programs