Proof Techniques for Operational Semantics

Meeting 6, CSCI 5535, Spring 2009

Announcements

- Homework 0 is graded (post tonight)
- Homework 1 is due today
- Office hours:
  - M 5:30pm-6:30pm (after class)
  - W 1:30pm-2:30pm

Review of Small-Step Operational Semantics

What is the key feature of small-step operational semantics?

- Why choose small-step versus big-step operational semantics?
  - Specify order of evaluation
  - Intermediate states
  - Small-step \( e, o \rightarrow e', o' \)
  - Big-step \( e, o \downarrow \eta \)

Quick Summary

- **Small-step operational semantics** models intermediate states
  - Execution is modeled as a (possible infinite) sequence of states
  - Not quite as easy as large-step natural semantics, though
- **Contextual operational semantics** is a small-step operational semantics where the atomic execution step is a rewrite of the program
Small-Step View of Evaluation

- A sequence of atomic rewrites:
  \[ x + (7-3), \sigma \rightarrow x + (4), \sigma \rightarrow 5 + 4, \sigma \rightarrow 9, \sigma \]
  \[ \sigma(x) = 5 \]

- Contextual view:
  
  If \( <r, \sigma> \rightarrow <e, \sigma'> \)
  
  then \( <H[r], \sigma> \rightarrow <H[e], \sigma'> \)

  \( r \) : redex
  \( H \) : context ("exp with a hole")

Context Decomposition Theorem

- If \( c \) is not "skip" then there exist unique \( H \) and \( r \) such that \( c \) is \( H[r] \)
  
  - "Exist" means progress
  
  - "Unique" means determinism

Redex or Not?

Decomposition? Reduction?

- skip
  
  \[ \text{No} \]
  
  \( c = H[r] \)

- skip; \( x := 0 \)
  
  \[ \text{Yes} \]
  
  \( H = \) skip
  
  \( r = x := 0 \)
  
  \[ \rightarrow x := 0 \]

Redex or Not?

Decomposition? Reduction?

- if true then \( x := -x \) else skip
  
  \[ \rightarrow x := -x \]

- if \( 5 \leq 0 \) then \( x := -x \) else skip
  
  \[ \text{No} \]
  
  \( H = \) if \( \text{true} \) then \( \ldots \)
  
  \( r = 5 \leq 0 \)
  
  \[ \rightarrow \text{false} \]

Redex or Not?

Decomposition? Reduction?

- if \( x \leq 0 \) then \( x := -x \) else skip
  
  \[ \text{No} \]
  
  \( H = \) if \( \cdot \leq 0 \) then \( \ldots \)
  
  \( r = (x, 0) \rightarrow (\sigma(x), \sigma) \)
  
  \( \rightarrow x := 0 \)

- while true do \( x := x + 1 \)
  
  \[ \text{Yes} \]

Redex or Not?

Decomposition? Reduction?

- while \( x \leq n \) do \( x := x + 1 \)
  
  \[ \text{Yes} \]
Summary: Contextual Operational Semantics

- Can view ● as representing the program counter
- The advancement rules for ● are non-trivial
  - At each step the entire command is decomposed
  - This makes contextual semantics inefficient to implement directly

- The major advantage of contextual semantics: allows a mix of local and global reduction rules
  - For IMP we have only local reduction rules: only the redex is reduced
  - Sometimes it is useful to work on the context too

Today's Plan

- Why Bother?
- Mathematical Induction
- Well-Founded Induction
- Structural Induction
  - "By induction on the structure of the derivation D"

Why Bother?

- I must convince you op. sem. proof techniques are useful.
- Recall class goals:
  - Understand PL research techniques and
  - Apply them to your research

Classic Example (Schema)

- "A well-typed program cannot go wrong."
  - Robin Milner
- When you design a new type system, you must show that it is safe (= that the type system is sound with respect to the operational semantics).
  - Preservation: "If you have a well-typed program and apply an op. sem. rule, the result is well-typed."
  - Progress: "a well-typed program will never get stuck in a state with no applicable op. sem. rules"
- Done for real languages: SML, SPARK ADA, Java
  - plus basically every toy PL research language ever
Instances

- CCured Project (Berkeley)
  - A program that is instrumented with CCured run-time checks ("adheres to the CCured type system") will not segfault (= "the x86 op. sem. rules will never get stuck").
- Vault Language (Microsoft Research)
  - A well-typed Vault program does not leak any tracked resources and invokes tracked APIs correctly (e.g., handles IRQL correctly in asynchronous Windows device drivers; cf. Capability Calculus).
- RC - Reference-Counted Regions For C (Intel Research)
  - A well-typed RC program gains the speed and convenience of region-based memory management but need never worry about freeing a region too early (run-time checks).
- Typed Assembly Language (Cornell)
  - Reasonable C programs (e.g., device drivers) can be translated to TALx86. Well-typed TALx86 programs are type- and memory-safe.

Induction!

- Probably most important technique for studying the formal semantics of PLs
  - To perform or understand PL research, you must grok this!
- Mathematical Induction (simple)
- Well-Founded Induction (general)
- Structural Induction (widely used in PL)

Mathematical Induction

- Goal: prove \( \forall n \in \mathbb{N}. P(n) \)
- **Base Case**: prove \( P(0) \)
- **Inductive Step**:
  - Prove \( \forall n. P(n) \Rightarrow P(n+1) \)
  - "Pick arbitrary \( n \), assume \( P(n) \), prove \( P(n+1) \)"
- Why does induction work?

Why does it work?

- There are no infinite descending chains of natural numbers
- For any \( n \), \( P(n) \) can be obtained by starting from the base case and applying \( n \) instances of the inductive step

Example (With IMP Eval. Semantics)

- Prove that if \( \sigma(x) \leq 6 \) then
  \[ \text{while } x \leq 5 \text{ do } x := x + 1, \sigma \Downarrow \sigma[x := 6] \]
- Reformulate the claim:
  - Let \( W = \text{while } x \leq 5 \text{ do } x := x + 1 \)
  - Let \( \sigma_0 = \sigma[x := 6 - i] \)
  - Claim: \( \forall i \in \mathbb{N} \). \( \langle W, \sigma_0 \Downarrow \rangle \)
- Now looks provable by mathematical induction on \( i \)

Evaluation Example (Base Case)

- Base case: \( i = 0 \) or \( \langle W, \sigma_0 \Downarrow, \sigma_0 \rangle \)
  - To prove an evaluation judgment, construct a derivation tree:
    \[ \sigma_0(x) = 6 \]
    \[ x, \sigma_0 \Downarrow, 6 \leq 5, \sigma_0 \Downarrow \text{ false} \]
    \[ x \leq 5, \sigma_0 \Downarrow \text{ false} \]
    \[ \text{while } x \leq 5 \text{ do } x := x + 1, \sigma_0 \Downarrow \sigma_0 \]
    \[ \sigma_0 = \sigma[x := 6 - i] \]
Evaluation Example (Inductive Case)

- Must prove \( \forall i \in \mathbb{N}. \prec \sigma_i \succ \Rightarrow \sigma_0 \)
- Pick an arbitrary \( i \in \mathbb{N} \)
- Assume that \( \prec \sigma_i \succ \Rightarrow \sigma_0 \)
- Now prove that \( \prec \sigma_{i+1} \succ \Rightarrow \sigma_0 \)
- Must construct a derivation tree:

\[
s_i = \sigma[x := 6 - i]
\]

- while \( x \leq 5 \) do \( x := x + 1 \)

Well-Founded Induction

- A relation \( \prec \subseteq \mathbb{A} \times \mathbb{A} \) is well-founded if there are no infinite descending chains in \( \mathbb{A} \)
  - Example: \( \prec \subseteq \mathbb{N} \times \mathbb{N} \) with \( (x, y) \prec \) \( x \prec y \)
    - aka the predecessor relation
    - Example: \( \prec \subseteq \mathbb{N} \times \mathbb{N} \) and \( x \neq y \)

  - Well-founded induction:
    - To prove \( \forall x \in \mathbb{A}. P(x) \) it is enough to prove \( \forall x \in \mathbb{A}. (\forall y \prec x \Rightarrow P(y)) \Rightarrow P(x) \)
    - If \( \prec \) is \( \prec \subseteq \mathbb{N} \times \mathbb{N} \) then we obtain mathematical induction as a special case

Well-Founded Induction: Examples

- Consider \( \prec \subseteq \mathbb{Z} \times \mathbb{Z} \) with \( (x, y) \prec \) \( xy < 0 \Rightarrow y = x - 1 \)
  - Induction principle: \( P(0) \land \forall x \leq 0. P(x) \Rightarrow P(x - 1) \land \forall x \geq 0. P(x) \Rightarrow P(x + 1) \)

- Consider \( \prec \subseteq (\mathbb{N} \times \mathbb{N}) \times (\mathbb{N} \times \mathbb{N}) \) and \( (x_1, y_1) \prec (x_2, y_2) \) iff \( x_1 \neq x_2 \lor (x_1 = x_2 \land y_1 \leq y_2 \land y_1 \neq y_2 + 1) \)
  - Induction principle: \( P(0,0) \land \forall x,y. P(x,y) \Rightarrow P(x + 1,y) \land P(x,y + 1) \)

  This has a common name. Anyone see?

Structural Induction (on Expressions)

- For \( e ::= n \mid x \mid e_1 + e_2 \mid e_1 \ast e_2 \)
  - Define \( \prec \subseteq A_{\text{exp}} \times A_{\text{exp}} \) such that
    - \( e_1 \prec e_2 \) if \( e_1 \neq e_2 \)
    - \( e_1 \prec e_2 \) if \( e_1 \neq e_2 \)
    - \( e_1 \prec e_2 \) if \( e_1 \neq e_2 \)
    - no other elements of \( A_{\text{exp}} \times A_{\text{exp}} \) are related by \( \prec \)
  - To prove \( \forall e \in A_{\text{exp}}. P(e) \)
    - prove \( \forall n \in \mathbb{Z}. P(n) \)
    - prove \( \forall x \in \mathbb{L}. P(x) \)
    - prove \( \forall e_1, e_2 \in A_{\text{exp}}. P(e_1) \land P(e_2) \Rightarrow P(e_1 \ast e_2) \)
    - prove \( \forall e_1, e_2 \in A_{\text{exp}}. P(e_1) \land P(e_2) \Rightarrow P(e_1 \ast e_2) \)

Example Proof Using Induction on the Structure of Expressions

- Let \( L(e) \) be the # of literals and variable occurrences in \( e \)
  - \( O(e) \) be the # of operators in \( e \)
  - Prove that \( \forall e \in A_{\text{exp}}. L(e) = O(e) + 1 \)
  - Proof: By induction on the structure of \( e \).
    - Case \( e = n \): \( L(n) = 1 \land O(n) = 0 \)
    - Case \( e = x \): \( L(x) = 1 \land O(x) = 0 \)

Notes on Structural Induction

- Called structural induction because the proof is guided by the structure of the expression
- One proof case per form of expression
  - Atomic expressions (with no subexpressions) are all base cases
  - Composite expressions are the inductive cases
- Structural induction is the most useful form of induction in the study of PL
Example Proof Using Induction on the Structure of Expressions

- Case \( e = e_1 + e_2 \):
  \[
  L(e) = L(e_1) + L(e_2) \\
  O(e) = O(e_1) + O(e_2) + 1
  \]
  By induction hypothesis, \( L(e_1) = O(e_1) + 1 \) and \( L(e_2) = O(e_2) + 1 \)

- Case \( e = e_1 \cdot e_2 \):
  \[
  L(e) = L(e_1) + L(e_2) \\
  O(e) = O(e_1) + O(e_2) + 1
  \]
  By induction hypothesis, \( L(e_1) = O(e_1) + 1 \) and \( L(e_2) = O(e_2) + 1 \) 
  Thus, \( L(e) = O(e_1) + O(e_2) + 2 = O(e) + 1 \)

“Try it at home!”

- Most proofs for the Aexp sublanguage of IMP can work by structural induction
- Small-step and big-step semantics obtain equivalent results:
  \[
  \forall e \in \text{Aexp. } \forall n \in \mathbb{Z}. \ e \rightarrow^* n \iff e \Downarrow n
  \]

“Obvious, right?”

- You are given a concrete state \( \sigma \).
  You have \( <x + 1, \sigma> \Downarrow 5 \)
  You also have \( <x + 1, \sigma> \Downarrow 88 \)
  Is this possible?

Let’s make sure

- Prove that IMP is deterministic
  \[
  \forall e \in \text{Aexp. } \forall n \in \mathbb{Z}. \ e \rightarrow^* n \iff e \Downarrow n
  \]
  \[
  \forall b \in \text{Bexp. } \forall n \in \mathbb{Z}. \ b \rightarrow^* n \iff b \Downarrow n
  \]
  \[
  \forall c \in \text{Com. } \forall \sigma, \sigma', \sigma'' \in \Sigma. \ c, \sigma \Downarrow c' \iff c, \sigma \Downarrow c' \iff c, \sigma' \Downarrow c'
  \]
  No immediate way to use mathematical induction
- For commands we cannot use induction on the structure of the command
  - While’s evaluation does not depend only on the evaluation of its strict subexpressions
    \[
    <b, \sigma> \Downarrow \text{true} \iff <c, \sigma> \Downarrow c' \iff <\text{while } b \text{ do } c, \sigma> \Downarrow \sigma'
    \]

We need something new!

- Some more powerful form of induction
  - With all the bells and whistles!
Recall Proof Systems

- Operational semantics assigns meanings to programs by listing rules of inference that allow to prove judgments by constructing derivations.
- A derivation is a tree-structured object made up of valid instances of inference rules.

Induction on the Structure of Derivations

- Key idea: The hypothesis does not just assume \( c \in \text{Com} \) but the existence of a derivation of \( <c, \sigma> \Downarrow <\sigma'> \).
- Derivation trees are also defined inductively, just like expression trees.
- A derivation is built of subderivations:

\[
\begin{align*}
\text{If } D_1 &:: <c, \sigma> \Downarrow \sigma' \\
\text{and } D_2 &:: <c, \sigma'> \Downarrow \sigma'', \text{ then } \sigma'' = \sigma'.
\end{align*}
\]

Notation: Naming Derivations

- Write \( D :: \text{Judgment} \) to mean “\( D \) is the derivation that proves \( \text{Judgment} \).”
- Example:

\[
D :: <e_1 + e_2, \sigma> \Downarrow n_1 + n_2
\]

Proving Com Evaluation is Deterministic

- Case

\[
D :: \begin{array}{c}
\text{skip, } \sigma \\
\text{Last rule used in } D \text{ was the one for skip}
\end{array}
\]

- This means that \( c = \text{skip} \) and \( \sigma' = \sigma \).
- By inversion, \( D :: <c, \sigma> \Downarrow \sigma' \) uses the rule for skip.
- Thus, \( \sigma'' = \sigma \).

- This is a base case in the induction.
If \( D :: \{ c, a \} \uplus a' \) and \( D :: \{ c, a \} \uplus a'' \), then \( a' = a'' \).

- **Case**
  
  \( D :: \begin{array}{c}
  D_1 :: \{ c_1, a \} \uplus a_1 \\
  D_2 :: \{ c_2, a \} \uplus a' \\
  \{ c', c_2, a \} \uplus a''
  \end{array} \)

  - By inversion, \( D :: \{ c, a \} \uplus a' \) uses the rule for sequencing and has subderivations
  
  \( D'_1 :: \{ c, a \} \uplus a'_1 \) and \( D'_2 :: \{ c_2, a \} \uplus a'' \)

  - By induction hypothesis on \( D_1 \) (with \( D'_1 \)); \( a_1 = a'_1 \)

  - By induction hypothesis on \( D_2 \) (with \( D'_2 \)); \( a'' = a'' \)

  This is a simple inductive case.

Summary: Induction on Derivations

- If you must prove \( \forall x \in A, P(x) \Rightarrow Q(x) \)
  
  - with A inductively defined and \( P(x) \) rule-defined
  
  - we pick arbitrary \( x \in A \) and \( D :: P(x) \)

  - we could do induction on both facts
    - \( x \in A \) leads to induction on the structure of \( x \)
    - \( D :: P(x) \) leads to induction on the structure of \( D \)

  - Generally, the induction on the structure of the derivation is more powerful and a safer bet

- Sometimes there are many choices for induction
  
  - choosing the right one is a trial-and-error process
  
  - a bit of practice can help a lot

- Point of reference for other semantics
  
  - Basis for much reasoning about programs
  
  - Often not compositional (see while)

- Basis for many proofs about a language
  
  - Especially when combined with type systems!

- Basis for much reasoning about programs
  
  - Simple and abstract (vs. implementations)

  - no low-level details such as stack and memory management, data layout, etc.

- No error conditions (sometimes implicitly, by rule applicability: "no applicable rule" = "get stuck")

- Precise specification of dynamic semantics

  - order of evaluation (or that it doesn’t matter)

Summary: Operational Semantics

- With A inductively defined and P(x) rule-defined

  - we pick arbitrary x ∈ A and D :: P(x)

  - we could do induction on both facts
    - x ∈ A leads to induction on the structure of x
    - D :: P(x) leads to induction on the structure of D

  - Generally, the induction on the structure of the derivation is more powerful and a safer bet

- Sometimes there are many choices for induction
  
  - choosing the right one is a trial-and-error process
  
  - a bit of practice can help a lot

- Point of reference for other semantics