CMSC 631 — Program Analysis and Understanding
Fall 2007

Induction and Operational Semantics

Formal Program Semantics

• Program analysis is often concerned with proving or disproving properties of a program

• How do we know that the facts we prove are correct?

• First, we need to understand a program’s semantics; that is, what it means
  • Then we can understand if the property holds for some set of programs
This Lecture

- Semantics of simple arithmetic expressions
  - “small-step” and “big-step”
  - Properties: determinacy and normal forms
  - Proposition of equivalence (without proof)
- The bread and butter of PL proofs: induction
  - Inductive definitions and structural induction
  - Evaluations as trees, amenable to inductive proofs
- Semantics of a more realistic language, with control flow

Arithmetic Expressions

- $x, y, z \in \text{Var}$
- $i, j, k \in \text{Int}$
- $e \in \text{Exp ::= x | i | } e_1 + e_2 | e_1 * e_2$

- Examples
  - $(x + 1) * y$
  - $(1 + 3) * (x * y)$
  - Etc.
Semantics of Arithmetic Expressions

- Defined as transitions between abstract machine states \((e,s)\)
  - \(e\) is the expression to evaluate
  - \(s\) is the program memory
- Formally:
  - \(\text{Store} = \text{Var} \rightarrow \text{Int}\)
    - (a function from variables to integers)
  - \(\text{MachineState} = \text{Exp} \times \text{Store}\)
  - \(\text{Transition} = \text{MachineState} \times \text{MachineState}\)
    - We write \(\text{MS} \rightarrow \text{MS'}\) to say that \((\text{MS}, \text{MS'})\) \(\in\) relation \(\rightarrow\)

Examples

- \((3+4,s) \rightarrow (7,s)\)
- \((2*6,s) \rightarrow (12,s)\)
- \((x,s) \rightarrow (s(x),s)\)

- The number of possible transitions is infinite (Why?)
  - Need a way to specify all possible members of the transition relation
- Solution: inference rules
Transition Relation: Computation

- **(var)** $s(x) = k$  
  
  $(x,s) \rightarrow (k,s)$

- **(plus)** $(i+j,s) \rightarrow (k,s)$  
  (where $k$ is the sum of $i$ and $j$)

- **(times)** $(i\ast j,s) \rightarrow (k,s)$  
  (where $k$ is the product of $i$ and $j$)

- These rules are sufficient to define the transitions we say on the previous slide
  - But what about compound expressions, like $(i+(j+k),s)$?

---

Transition Relation: Congruence

- **(left plus)** $\frac{(e_1,s) \rightarrow (e'_1,s)}{(e_1+e_2,s) \rightarrow (e'_1+e_2,s)}$

- **(right plus)** $\frac{(e_2,s) \rightarrow (e'_2,s)}{(i+e_2,s) \rightarrow (i+e'_2,s)}$

- **(left times)** $\frac{(e_1,s) \rightarrow (e'_1,s)}{(e_1\ast e_2,s) \rightarrow (e'_1\ast e_2,s)}$

- **(right times)** $\frac{(e_2,s) \rightarrow (e'_2,s)}{(i\ast e_2,s) \rightarrow (i\ast e'_2,s)}$
Transitions as Derivations

• Rules define derivations (proofs) that a given state transitions to another state

• Example

\[
\begin{align*}
\text{(var)} & \quad s(x) = 21 \\
\text{(right times)} & \quad (x,s) \rightarrow (21,s) \\
\text{(right plus)} & \quad (2\times x,s) \rightarrow (2\times 21,s) \\
& \quad (12+2\times x,s) \rightarrow (12+2\times 21,s)
\end{align*}
\]

• But how can we discover such a proof?
  • How can we determine \((MS \rightarrow MS')\) given \(MS\)?

Proof by Goal-directed Search

• Compare the structure of a machine state \((e,s)\) to the conclusion of each transition rule
  • Then apply the rule that matches

• Example
  • \((12+(2\times x),s) \rightarrow ???\)
  • Has form \((i+e2,s)\)
    - Where \(i = 12\) and \(e2 = 2\times x\).
  • Thus matches the rule \((right-plus)\).
  • Then we apply the same approach to proving the premises.
Evaluation

• We are interested in the “final result” of a given arithmetic expression. We call this result a value.
  • The process of reducing an expression to a value is thus called evaluation.

• Define
  • \( v \in \text{Values} = \text{Int} \)
  • \( \rightarrow^* \) as the reflexive, transitive closure of \( \rightarrow \)
    - \( MS \rightarrow^* MS \) for all \( MS \)
    - \( MS \rightarrow^* MS' \) and \( MS' \rightarrow MS'' \) imply \( MS \rightarrow^* MS'' \)

• Can recursively apply goal-directed search for each transition

Example Evaluation

- \( s(x) = 21 \)
- \( (\text{var}) \quad \frac{s(x) = 21}{(x,s) \rightarrow (21,s)} \)
- \( (\text{right times}) \quad \frac{(x,s) \rightarrow (21,s)}{(2 \times x,s) \rightarrow (2 \times 21, s)} \)
- \( (\text{right plus}) \quad \frac{(2 \times 21, s) \rightarrow (12 + 2 \times 21, s)}{(12 + 2 \times 21, s) \rightarrow (12 + 42, s)} \)

and

- \( (\text{times}) \quad \frac{(2 \times 21, s) \rightarrow (42, s)}{(12 + 2 \times 21, s) \rightarrow (12 + 42, s)} \)

implies

- \( (12 + 2 \times x, s) \rightarrow^* (12 + 42, s) \)
Properties of $\rightarrow$

- **Determinacy**: Given a machine state $(e,s)$, such that $\text{fv}(e) \subseteq \text{dom}(s)$, then either $e$ is an integer, or else there exists at most one $e'$ such that $(e,s) \rightarrow (e',s)$.
  - Proof: by induction on the structure of $e$.

- **Confluence**: if $(e,s) \rightarrow^* (e_1,s)$ and $(e,s) \rightarrow^* (e_2,s)$ then there exists some $e_3$ such that $(e_1,s) \rightarrow^* (e_3,s)$ and $(e_2,s) \rightarrow^* (e_3,s)$.
  - Proof: follows from Determinacy.

Digression: Proof by Induction

- When trying to prove property $P(n)$ where $n$ is a natural number, it suffices to prove that
  - $P(0)$ holds and $P(n)$ implies $P(n+1)$
    - Called natural number induction principle
  - $P(0)$ and $P(1)$ and ... $P(n)$ implies $P(n+1)$
    - Called complete induction on natural numbers

- Example:
  - Let $\text{sum}(n) = \text{sum of all numbers between 1 and } n$
  - Prove $P(n) \equiv \forall m. m < n \text{ implies } \text{sum}(m) < \text{sum}(n)$
Proof

- **Base case: \( n = 0 \)**
  - \( P(0) \) holds vacuously because there is no natural number \( m \) such that \( m < 0 \).

- **Inductive case: \( n > 0 \)**
  - Given \( P(n) \equiv \forall m. \ m < n \implies \text{sum}(m) < \text{sum}(n) \)
  - To prove \( P(n+1) \), suffices to prove \( \text{sum}(n) < \text{sum}(n+1) \)
    - \( \text{sum}(n+1) = n+1 + \text{sum}(n) \)
    - \( \text{sum}(n+1) > \text{sum}(n) \) since \( n \) is positive

Inductive Definitions

- The natural numbers \( \text{num} \) are an inductively-defined set
  - \( Z \in \text{num} \)
  - \( n \in \text{num} \implies (S \ n) \in \text{num} \)
    - (read \( S \ n \) as "successor of \( n \)", i.e., \( n+1 \))

- Proofs proceed by considering each case of the inductive definition
  - Prove \( P(Z) \) and \( P(n) \Rightarrow P(S \ n) \), or
  - Prove \( P(Z) \land P(S \ Z) \land \ldots \land P(n) \Rightarrow P(S \ n) \)
    - Follows inductive structure of \( \text{num} \)
Proofs on trees

• \( e ::= x \mid i \mid e_1 + e_2 \mid e_1 \ast e_2 \)

• Abstract syntax trees are inductive definitions
  - \( x \in \exp \)
  - \( i \in \exp \)
  - \( \forall e_1, e_2. (e_1 \in \exp \text{ and } e_2 \in \exp) \Rightarrow (e_1 + e_2) \in \exp \)
  - \( \forall e_1, e_2. (e_1 \in \exp \text{ and } e_2 \in \exp) \Rightarrow (e_1 \ast e_2) \in \exp \)

• Can prove by induction on the structure of the tree, following each form of the definition

Simple Property of expressions

• \( \text{Size} : \exp \rightarrow \text{int} \)
  - \( \text{Size}(x) = 1 \)
  - \( \text{Size}(i) = 1 \)
  - \( \text{Size}(e_1 + e_2) = \text{Size}(e_1) + \text{Size}(e_2) \)
  - \( \text{Size}(e_1 \ast e_2) = \text{Size}(e_1) + \text{Size}(e_2) \)

• \( \text{Consts} : \exp \rightarrow \mathcal{P}(\text{int}) \)
  - \( \text{Consts}(x) = \emptyset \)
  - \( \text{Consts}(i) = \{i\} \)
  - \( \text{Consts}(e_1 + e_2) = \text{Consts}(e_1) \cup \text{Consts}(e_2) \)
  - \( \text{Consts}(e_1 \ast e_2) = \text{Consts}(e_1) \cup \text{Consts}(e_2) \)

• Prove: \( |\text{Consts}(e)| \leq \text{Size}(e) \)
Proof by Structural Induction

- Prove: \( P(e) = |\text{Consts}(e)| \leq \text{Size}(e) \)
  - Induction principle: may assume \( P(e') \) for all immediate subterms \( e' \) of \( e \)
    - Corresponds to natural number induction principle
  - Case: \( e = x \)
    - \( |\text{Consts}(x)| = 0, \text{Size}(x) = 1, 0 \leq 1 \)
  - Case: \( e = i \)
    - \( |\text{Consts}(i)| = 1, \text{Size}(i) = 1, 1 \leq 1 \)
  - Case: \( e = e_1 + e_2 \)
    - \( |\text{Consts}(e_1 + e_2)| = |\text{Consts}(e_1)| + |\text{Consts}(e_2)| \)
      - \( \text{Size}(e_1 + e_2) = \text{Size}(e_1) + \text{Size}(e_2) \)
      - By induction, \( |\text{Consts}(e_1)| \leq \text{Size}(e_1) \), likewise for \( e_2 \)
      - Hence \( |\text{Consts}(e_1 + e_2)| \leq |\text{Consts}(e_1)| + |\text{Consts}(e_2)| \leq \text{Size}(e_1) + \text{Size}(e_2) \)
  - Case: \( e = e_1 \times e_2 \) similar

ML datatypes: inductive definitions

- type num =
  - Zero
  - | Succ of num

- (* sum from 1 to n *)
- let rec sum = function
  - Zero -> Zero
  - | Succ n -> add (Succ n) (sum n)

- let rec lt m n = ... (* m < n *)
Proof follows structure

- Consider our earlier proof of
  - $P(n) = \forall m. (lt m n)$ implies $lt (sum m) (sum n)$
- Each case is a branch in the def of $num$
  - $n = Zero$, vacuously true
  - $n = Succ n'$ for some $n'$
    - $sum (Succ n') = add (Succ n') (sum n')$
    - Thus $lt (sum n') (sum (Succ n'))$
    - $\forall m. (lt m n')$ implies $lt (sum m) (sum n')$
      - follows by induction
    - Result follows from these two facts and transitivity of $lt$

(Proof) derivations are trees too

- Each inference rule
  - $P_1 P_2 \ldots P_n$
    - $Q$
  - Can be read as (roughly)
    - $(P_1 \in D) \land (P_2 \in D) \land \ldots (P_n \in D) \Rightarrow Q \in D$

- Proofs once again follow the form of each variant, with one variant per rule
  - So each case of proof corresponds to a rule
  - (Conversely, can represent BNF as inference rules!)
Proofs on Evaluation Trees

**Determinacy**: Given a machine state \((e, s)\), such that \(\text{fv}(e) \subseteq \text{dom}(s)\), then either \(e\) is an integer, or else there exists at most one \(e'\) such that \((e, s) \rightarrow (e', s)\).

- Proof by induction on the structure of \((e, s)\)
- case \(e \equiv i\): follows directly
- case \(e \equiv x\): only rule (Var) applies, \(e' = s(x)\)
- case \(e \equiv e_1 + e_2\):
  - Apply induction on each of \(e_1\) and \(e_2\), and then show that only one rule applies, preserving determinacy
- (A different approach: Pierce p. 37)

In short

- Proofs by induction follow the inductive structure of the data they consider

- Further reading in Pierce
  - Induction principles on natural numbers: p. 19
  - Induction on terms (BNF trees): pps. 29-32
  - Foundational principles: pps. 282-284

- Back to the semantics of arithmetic expressions …
Recall our current semantics

(left plus) \[ (e_1, s) \rightarrow (e_1', s) \]
\[ (e_1 + e_2, s) \rightarrow (e_1'e_2, s) \]

(right plus) \[ (e_2, s) \rightarrow (e_2', s) \]
\[ (i + e_2, s) \rightarrow (i + e_2', s) \]

(left times) \[ (e_1, s) \rightarrow (e_1', s) \]
\[ (e_1 * e_2, s) \rightarrow (e_1'e_2, s) \]

(right times) \[ (e_2, s) \rightarrow (e_2', s) \]
\[ (i * e_2, s) \rightarrow (i * e_2', s) \]

Alternative Semantics

(left plus) \[ (e_1, s) \rightarrow (e_1', s) \]
\[ (e_1 + e_2, s) \rightarrow (e_1'e_2, s) \]

(right plus') \[ (e_2, s) \rightarrow (e_2', s) \]
\[ (e_1 + e_2, s) \rightarrow (e_1'e_2, s) \]

(left times) \[ (e_1, s) \rightarrow (e_1', s) \]
\[ (e_1 * e_2, s) \rightarrow (e_1'e_2, s) \]

(right times') \[ (e_2, s) \rightarrow (e_2', s) \]
\[ (e_1 * e_2, s) \rightarrow (e_1'e_2, s) \]
Determinacy Lost

Given \((y+2\times x, s)\), we now have either

\[
\begin{align*}
\text{(var)} & \quad s(x) = 21 \\ 
\text{(right times)} & \quad (x,s) \rightarrow (21,s) \\ 
\text{(right plus)} & \quad (2\times x, s) \rightarrow (2\times 21, s) \\ 
\quad & \quad (y+2\times x, s) \rightarrow (y+2\times 21, s)
\end{align*}
\]

or

\[
\begin{align*}
\text{(var)} & \quad s(y) = 12 \\ 
\text{(left plus)} & \quad (y,s) \rightarrow (12,s) \\ 
\quad & \quad (y+2\times x, s) \rightarrow (12+2\times x, s)
\end{align*}
\]

Confluence Retained

• We can still prove Confluence for the new semantics.
  • The proof is harder. We prove a simpler lemma first, that shows we have confluence for \(\rightarrow\), then we repeatedly apply this lemma for \(\rightarrow^*\).

• Normal Forms: if \((e, s) \rightarrow^* (i, s)\) and \((e, s) \rightarrow^* (j, s)\) then \(i = j\).
  • Follows from Confluence.
  • Thus there is a unique meaning for each expression \(e\), which is its normal form.
Big-Step Semantics

• If we are really concerned only with normal forms, we might want to dispense with intermediate states.

• Define a relation \((e, s) \Downarrow i\)
  • Read: expression \(e\) evaluates to \(i\) in state \(s\)

Big-Step Inference Rules

\[
\begin{align*}
\text{(var)} & & s(x) = k & & (x, s) \Downarrow k \\
\text{(id)} & & (i, s) \Downarrow i \\
\text{(plus)} & & (e_1, s) \Downarrow i & & (e_2, s) \Downarrow j & & (e_1 + e_2, s) \Downarrow k \\
& & & & & & \text{(where } k \text{ is the sum of } i \text{ and } j) \\
\text{(times)} & & (e_1, s) \Downarrow i & & (e_2, s) \Downarrow j & & (e_1 \cdot e_2, s) \Downarrow k \\
& & & & & & \text{(where } k \text{ is the product of } i \text{ and } j)
\end{align*}
\]
Equivalence of the Semantics

- Define \( \text{eval}((e,s),i) \) iff \( (e,s) \rightarrow^* (i,s) \).
- **Equivalence**: \( \text{eval}((e,s),i) \) iff \( (e,s) \Downarrow i \).
  - Proof: Must show \( (e,s) \Downarrow i \) implies \( \text{eval}((e,s),i) \) and \( \text{eval}((e,s),i) \) implies \( (e,s) \Downarrow i \).
  - The first is by induction on the structure of the derivation \( (e,s) \Downarrow i \).
  - The second is comes as a corollary of the lemma: if \( (e,s) \rightarrow^n (e',s) \) and \( (e',s) \Downarrow i \), then \( (e,s) \Downarrow i \).
    - where \( (e,s) \rightarrow^n (e',s) \) means
    - \( (e,s) \rightarrow (e_1,s) \rightarrow \ldots \rightarrow (e_{n-1},s) \rightarrow (e',s) \)
    - The proof is by induction on the length \( n \)

A Language of Commands

- Extend the language to include commands \( c \)
  - \( c ::= \)
  - skip
  - \( x := e \)
  - \( \text{if}_{\not=0} e \) then \( c_1 \) else \( c_2 \)
  - \( \text{while}_{\not=0} e \) do \( c \)
  - \( c_1; c_2 \)
Semantics of Commands

- Two small-step relations; normal form:
  - \((c,s) \rightarrow s'\)
  - \((c,s) \rightarrow (c',s')\)

- Reflexive, transitive closure of both relations:
  - \((c,s) \rightarrow^* s'\)

- One big-step relation
  - \((c,s) \Downarrow s'\)

Small-step Computation Rules

- \((\text{assign})\) \((x := i,s) \rightarrow s[i\backslash x]\)
- \((\text{skip})\) \((\text{skip},s) \rightarrow s\)
- \((\text{if}0)\) \((\text{if}_{\not=0} 0 \text{ then } c_1 \text{ else } c_2,s) \rightarrow (c_2,s)\)
- \((\text{if}n)\) \((\text{if}_{\not=0} i \text{ then } c_1 \text{ else } c_2,s) \rightarrow (c_1,s)\) (where \(i \not= 0\))
Semantics of While

(while)

\[(\text{while}_{\neg \text{not}0} e \text{ do } c, s) \rightarrow (\text{if}_{\neg \text{not}0} e \text{ then } (c; \text{ while}_{\neg \text{not}0} e \text{ do } c) \text{ else } \text{skip}, s)\]

• Semantics by “unrolling” the loop once and evaluating that
• Different from other rules: not in terms of sub-components

Small-step Congruence Rules

(cong1)
\[
\frac{(e,s) \rightarrow (e',s)}{(x := e,s) \rightarrow (x := e',s)}
\]

(cong2)
\[
\frac{(e,s) \rightarrow (e',s)}{(\text{if}_{\neg \text{not}0} e \text{ then } c_1 \text{ else } c_2, s) \rightarrow (\text{if}_{\neg \text{not}0} e' \text{ then } c_1 \text{ else } c_2, s)}
\]

(seq-l)
\[
\frac{(c_1,s) \rightarrow (c_1',s')}{(c_1; c_2, s) \rightarrow (c_1'; c_2, s')}\]

(seq-r)
\[
\frac{(c_1,s) \rightarrow s'}{(c_1; c_2, s) \rightarrow (c_2, s')}\]
Big-step Rules

\[
\begin{align*}
\text{(assign)} & \quad (e,s) \downarrow i \\
\text{(seq)} & \quad (x := e,s) \downarrow s[x] \\
\text{(skip)} & \quad (\text{skip},s) \downarrow s \\
\text{(if0)} & \quad (c_1,s) \downarrow s' \\
\text{(ifn)} & \quad (\text{if not0 e then } c_1 \text{ else } c_2,s) \downarrow s' (\text{where } i \neq 0)
\end{align*}
\]

Semantics of while

\[
\begin{align*}
\text{(while)} & \quad (\text{if not0 e then } (c; \text{while not0 e do } c) \text{ else skip},s) \downarrow s' \\
\text{(while)} & \quad (\text{while not0 e do } c,s) \downarrow s'
\end{align*}
\]

• Similar to small-step while, but we evaluate the expansion in the premise
Non-terminating Programs

- If the meaning of a command is its final store, what is the meaning of non-terminating program?
  - Nothing!
  - Non-terminating programs have infinitely-sized derivation trees, and thus do not appear in the relation.

- Small step semantics is still useful, since we can prove things about intermediate computations.

Hybrid Semantics

- It is convenient to combine the small and big-step semantics
  - Take big-steps for expressions
  - Take small steps for commands (could have side-effects)

- Will make it simpler to set up the inductive hypothesis in our proof.
Hybrid Rules

\[
\begin{align*}
\text{(assign)} & \quad (x := e, s) \rightarrow s[x := e] \\
\text{(skip)} & \quad \text{(skip, s)} \rightarrow s \\
\text{(if0)} & \quad (e, s) \downarrow (0, s) \\
& \quad (\text{if} \not= 0 e \text{ then } c_1 \text{ else } c_2, s) \rightarrow (c_2, s) \\
\text{(ifn)} & \quad (e, s) \downarrow (i, s) \\
& \quad (\text{if} \not= 0 e \text{ then } c_1 \text{ else } c_2, s) \rightarrow (c_1, s)
\end{align*}
\]

Semantics of while

\[
\begin{align*}
\text{(while0)} & \quad (e, s) \downarrow (0, s) \\
& \quad (\text{while} \not= 0 e \text{ do } c, s) \rightarrow s \\
\text{(whilen)} & \quad (e, s) \downarrow (i, s) \\
& \quad (\text{while} \not= 0 e \text{ do } c, s) \rightarrow (c; \text{while} \not= 0 e \text{ do } c, s)
\end{align*}
\]

(\text{where } i \not= 0)
What are these semantics good for?

• We started down this path for a reason:
  ▪ How can I prove that a particular property holds for my program

• Possible properties
  ▪ Determinacy and confluence (today)
  ▪ Well-typedness (next time)
  ▪ Equivalence
  ▪ Correctness
  ▪ Termination