Operational Semantics

CMSC 631 — Program Analysis and Understanding
Fall 2006

Formal Program Semantics

• Data flow analysis defines a set of transfer functions that aim to prove facts about 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

Today’s Lecture

• Semantics of simple arithmetic expressions
  • “small-step” and “big-step”
  • Properties: determinacy and normal forms
  • Proposition of equivalence (without proof)

• Semantics of a more realistic language, with control flow, like what we used for data flow analysis

• Statement of (partial) correctness for dataflow analysis
  • Proof sketch

Arithmetic Expressions

• x, y, z ∈ Var
• i, j, k ∈ Int
• e ∈ Exp ::= x | i | e_1 + e_2 | e_1 * e_2

• Examples
  • (x + 1) * y
  • (1 + 3) * (x * y)
  • Etc.

Examples

• (3 + 4, s) → (7, s)
• (2 * 6, s) → (12, s)
• (x, s) → (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

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:
  • Store = Var → Int
    • (a function from variables to integers)
  • MachineState = Exp × Store
  • Transition = MachineState × MachineState
    • We write MS → MS’ to say that (MS, MS’) ∈ relation

Arithmetic Expressions

• x, y, z ∈ Var
• i, j, k ∈ Int
• e ∈ Exp ::= x | i | e_1 + e_2 | e_1 * e_2

• Examples
  • (x + 1) * y
  • (1 + 3) * (x * y)
  • Etc.
Transition Relation: Computation

- **(var)** \( s(x) = k \) → \((k,s)\)
  - if \( s(x) = k \) then \((k,s)\) is a legal transition
- **(plus)** \((i+j,s) \rightarrow (k,s)\) (where \( k \) is the sum of \( i \) and \( j \))
- **(times)** \((i\cdot j,s) \rightarrow (k,s)\) (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 \((i+(i\cdot k),s)\)?

Transition Relation: Congruence

- **(left plus)** \((e_1 + e_2,s) \rightarrow (e_1' + e_2,s)\)
- **(right plus)** \((i + e_2,s) \rightarrow (i + e_2',s)\)
- **(left times)** \((e_1 \cdot e_2,s) \rightarrow (e_1' \cdot e_2,s)\)
- **(right times)** \((i \cdot e_2,s) \rightarrow (i \cdot e_2',s)\)

Transitions as Proofs

- Rules define proofs that a given machine state transitions to another machine state
- **Example**
  - \( (\var s(x) = 21) \) → \((21,s)\)
  - \( (\times) \quad (2\times x,s) \rightarrow (2\times 21,s)\)
  - \( (\plus) \quad (12+2\times x,s) \rightarrow (12+2\times 21,s)\)

- But how can we discover such a proof?
  - How can we determine \((\text{MS} \rightarrow \text{MS}')\) given \(\text{MS}\)?

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**
  - \((12+(2\times x),s)\) → ???
    - Has form \((\plus e_2, s)\)
      - Where \(i = 12\) and \(e_2 = 2\times x\).
    - Thus matches the rule \((\plus)\).
  - 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 \( \rightarrow \)
  - Can recursively apply goal-directed search for each transition

Example Evaluation

- \( (\var s(x) = 21) \) → \((21,s)\)
  - \( (\times) \quad (2\times x,s) \rightarrow (2\times 21,s)\)
  - \( (\plus) \quad (12+2\times x,s) \rightarrow (12+2\times 21,s)\)
  - \( (\times) \quad (2\times 21,s) \rightarrow (42,s)\)
  - \( (\plus) \quad (12+2\times 21,s) \rightarrow (12+42,s)\)

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

- **Determinacy**: Given a machine state $(e, s)$, 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.

Alternative Semantics

- $(e_1, s) \rightarrow (e_1', s)$
- $(e_1 + e_2, s) \rightarrow (e_1' + e_2, s)$
- $(e_1 \cdot e_2, s) \rightarrow (e_1' \cdot e_2, s)$

Determinacy Lost

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

- $(var) s(x) = 21$
- $(right \ times) (2 \cdot x, s) \rightarrow (21, s)$
- $(right \ plus) (y + 2 \cdot x, s) \rightarrow (y + 2 \cdot 21, s)$

or

- $(var) s(y) = 12$
- $(right \ times) (y + 2 \cdot x, s) \rightarrow (12 + 2 \cdot x, s)$

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

- $(var) s(x) = k$
- $(id) (i, s) \Downarrow i$
- $(plus) (e_1, s) \Downarrow i \quad (e_2, s) \Downarrow j \quad (e_1 + e_2, s) \Downarrow k$
  - $(where \ k \ is \ the \ sum \ of \ i \ and \ j)$
- $(times) (e_1, s) \Downarrow i \quad (e_2, s) \Downarrow j \quad (e_1 \cdot e_2, s) \Downarrow (k, s)$
  - $(where \ k \ is \ the \ product \ of \ i \ and \ j)$
**Equivalence of the Semantics**

- Define \( \text{eval}((e,s),i) \) iff \( (e,s) \rightarrow^* i \).

- **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 a corollary of the lemma: if \( (e,s) \rightarrow n (e',s) \) and \( (e',s) \downarrow i \), then \( (e,s) \downarrow i \).

  - The proof is by induction on the length \( n \).

**A Language of Commands**

- Extend the language to include commands \( c \)

  - \( c ::= \)
  - \( \text{skip} \)
  - \( x := e \)
  - \( \text{if} \neg 0 e \text{ then } c_1 \text{ else } c_2 \)
  - \( \text{while} \neg 0 e \text{ do } c \)
  - \( c_1; c_2 \)

**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**

- **Assign**
  \[ (\text{assign}) \quad \begin{array}{l}
  (x := e, s) \\
  \rightarrow \\
  s[x := e]
  \end{array} \]

- **Skip**
  \[ (\text{skip}) \quad \begin{array}{l}
  \text{(skip), s} \\
  \rightarrow \\
  s
  \end{array} \]

- **If True**
  \[ (i \text{if } 0 \text{ then } c_1 \text{ else } c_2, s) \\
  \rightarrow \\
  (c_2, s) \quad (\text{where } i \neq 0) \]

- **If False**
  \[ (i \text{if } 0 \text{ then } c_1 \text{ else } c_2, s) \\
  \rightarrow \\
  (c_1, s) \]

**Small-step Congruence Rules**

- **Congruence 1**
  \[ (\text{cong1}) \quad \begin{array}{l}
  (e,s) \rightarrow (e',s) \\
  (x := e, s) \\
  \rightarrow \\
  (x := e', s)
  \end{array} \]

- **Congruence 2**
  \[ (\text{cong2}) \quad \begin{array}{l}
  (e,s) \rightarrow (e',s) \\
  (\text{if } 0 \text{ then } c_1 \text{ else } c_2, s) \\
  \rightarrow \\
  (\text{if } 0 \text{ then } c_1 \text{ else } c_2, s)
  \end{array} \]

- **Seq 0**
  \[ (\text{seq0}) \quad \begin{array}{l}
  (c_1,s) \rightarrow (c_1',s') \\
  (c_1;c_2,s) \\
  \rightarrow \\
  (c_1';c_2,s')
  \end{array} \]

- **Seq 1**
  \[ (\text{seq1}) \quad \begin{array}{l}
  (c_1,s) \rightarrow s' \\
  (c_1;c_2,s) \\
  \rightarrow \\
  (c_2,s')
  \end{array} \]

**Semantics of While**

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

- \[ (\text{while}) \quad \begin{array}{l}
  (\text{while } 0 \text{ e do } c,s) \\
  \rightarrow \\
  (\text{if } 0 \text{ e then } (c; \text{while } 0 \text{ e do } c) \text{ else skip}, s)
  \end{array} \]
Big-step Rules

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

Semantics of while

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

• 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 \(\downarrow\) relation.

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

What are these semantics good for?

• We started down this path for a reason:
  • How can I prove that a particular dataflow analysis (say) is correct?

• Now that we know what programs mean, we can prove that facts we learn about them through analysis are actually sensible.

Hybrid Semantics

• For the purposes of this proof, 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

\[
\begin{align*}
\text{(assign)} & \quad (e,s) \downarrow (i,s) & \quad (\text{skip}) & \quad (\text{skip},s) \rightarrow s \\
\text{(if0)} & \quad (e,s) \downarrow (0,s) & \quad (c_2,s) \rightarrow (c_2,s) \\
\text{(ifn)} & \quad (e,s) \downarrow (i,s) & \quad (c_2,s) \rightarrow (c_1,s) \quad (\text{where } i \neq 0)
\end{align*}
\]
**Semantics of while**

\[
\begin{align*}
(\text{while} \ 0) & \Downarrow (0, s) \\
(\text{while} \ 0 \ e \ \text{do} \ c, s) & \rightarrow s \\
(\text{while} \ 0 \ e \ \text{do} \ c, s) & \rightarrow (c; \text{while} \ 0 \ e \ \text{do} \ c, s)
\end{align*}
\]

(where \( i \equiv 0 \))

---

**Example: Prove LV analysis is correct**

- **The live variable analysis** determined
  - For each command \( c \), those variables \( V \) for which there exists a path from \( c \) to a use of \( x \in V \) that does not assign to \( x \).
  - Call \( LV(c) \) the set of live variables at command \( c \).

- **What do we want to prove?**
  - Hint: we want to discover an invariant that is preserved by the small-step transition relation.

---

**Live Variables Analysis as Equations**

- **Kill**
  - \((x := e) = \{x\}\)
  - \((\text{skip}) = \emptyset\)
  - \((e) = \emptyset\)
- **Gen**
  - \((x := e) = \text{FV}(e)\)
  - \((\text{skip}) = \emptyset\)
  - \((e) = \text{FV}(e)\)

\[
\begin{align*}
LV_{\text{in}}(c) & = \emptyset \quad \text{if } \text{succ}(c) = \emptyset \\
LV_{\text{in}}(c) & = \bigcup LV_{\text{in}}(c') \quad \forall c' \in \text{succ}(c) \\
LV_{\text{out}}(c) & = \left( LV_{\text{out}}(c) \setminus \text{Kill}(c) \right) \cup \text{Gen}(c)
\end{align*}
\]

---

**Useful Lemmas**

- **Lemma**: the least solution of the constraint system and the least solution of the equation system coincide.
- **Lemma**: If \( live \) is a solution for some program \( c \), then \( live \) is a solution to all \( c' \) that are “sub-programs” of \( c \).
  - A sub-program has a subset of the statements and edges in the corresponding flow graph.
  - This implies that a correct solution is preserved during evaluation, since the flow graph either remains the same or shrinks.

---

**Make Equations into Constraints**

\[
\begin{align*}
LV_{\text{in}}(c) & \supseteq \emptyset \quad \text{if } \text{succ}(c) = \emptyset \\
LV_{\text{in}}(c) & \supseteq \bigcup LV_{\text{in}}(c') \quad \forall c' \in \text{succ}(c) \\
LV_{\text{out}}(c) & \supseteq \left( LV_{\text{out}}(c) \setminus \text{Kill}(c) \right) \cup \text{Gen}(c)
\end{align*}
\]

- Write \( live_{\text{in}} \) and \( live_{\text{out}} \) as solutions to these equations (maps commands to variable sets)
- Refer to both solutions together as \( live \)
- Write \( N \) and \( X \) as shorthand for \( live_{\text{in}} \) and \( live_{\text{out}} \)

---

**Proof by Induction**

- Need to construct the inductive hypothesis
  - When does the result of \( LV \) coincide with the actual evaluation of the program?
    - When the variables that are considered live are not assigned to
  - Define
    - \( s \sim s' \) if \( \forall x \in V. \ s(x) = s'(x) \)
  - States that two stores agree on the variables in \( V \)
    - In the proof, these will be the variables that should be live.
Correctness

- If live is a solution to LV for program \( c \) then
  - If \( (c,s_1) \rightarrow (c',s_1') \) and \( s_1 \not\sim_{N(init)} s_2 \) then \( \exists s_2' \) such that \( (c,s_2) \rightarrow (c',s_2') \) and \( s_1 \not\sim_{N(init)} s_2' \).
  - If \( (c,s_1) \rightarrow s_1' \) and \( s_1 \not\sim_{X(init)} s_2 \) then \( \exists s_2' \) such that \( (c,s_2) \rightarrow s_2' \) and \( s_1 \not\sim_{X(init)} s_2' \).

Corollary: If live is a solution for \( c \) then

- If \( (c,s_1) \rightarrow^\ast (c',s_1') \) and \( s_1 \not\sim_{N(init)} s_2 \) then \( \exists s_2' \) such that \( (c,s_2) \rightarrow^\ast (c',s_2') \) and \( s_1 \not\sim_{N(init)} s_2' \).
- If \( (c,s_1) \rightarrow^\ast s_1' \) and \( s_1 \not\sim_{X(init)} s_2 \) then \( \exists s_2' \) such that \( (c,s_2) \rightarrow^\ast s_2' \) and \( s_1 \not\sim_{X(init)} s_2' \).

- Proof by induction on length of the derivation sequence.

Sample case

- (assign)
  - \( (x := e,s_1) \rightarrow s_1[x] \) where \( (e,s_1) \Downarrow i \)
  - \( (x := e,s_2) \rightarrow s_2[x] \) where \( (e,s_2) \Downarrow j \)
  - \( N(c) = X(c) \setminus \{x\} \cup FV(e) \)
  - Thus
    - \( s_1 \not\sim_{N(i)} s_2 \) implies \( i = j \)
  - So, take \( s_1' = s_2[x] \) and we have \( s_1' \not\sim_{N(c)} s_2' \) as required.