Proof Techniques for Language Analysis

Lecture 3
ECS 240
Plan

• We’ll study various flavors of induction
  - mathematical induction
  - well-founded induction
  - structural induction
Induction

• Probably the single most important technique for the study of formal semantics of programming languages

• Of several kinds
  - mathematical induction (the simplest)
  - well-founded induction (the most general)
  - structural induction (the most widely used in PL)
Mathematical Induction

• **Goal:** prove that $\forall n \in \mathbb{N}. P(n)$

• **Strategy:** (2 steps)
  1. Base case: prove that $P(0)$
  2. Inductive case:
     - pick an arbitrary $n \in \mathbb{N}$
     - assume that $P(n)$ holds
     - prove that $P(n + 1)$
     - or, formally prove that $\forall n \in \mathbb{N}. P(n) \Rightarrow P(n+1)$
Mathematical Induction. Notes.

- The inductive case looks similar to the overall goal
  \[ \forall n \in \mathbb{N}. P(n) \Rightarrow P(n+1) \quad \text{vs.} \quad \forall n \in \mathbb{N}. P(n) \]
  - but it is simpler because of the assumption that \( P(n) \) holds

- Why does mathematical induction work?
  - The key property of \( \mathbb{N} \) is that there are no infinite descending chains of naturals. It has to stop somewhere.
  - For each \( n \), \( P(n) \) can be obtained from the base case and \( n \) uses of the inductive case
Example of Mathematical Induction

- Recall the evaluation rules for IMP commands
- 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 $\sigma_i = \sigma[x := 6 - i]$
  - Claim: $\forall i \in \mathbb{N}. <W, \sigma_i> \Downarrow \sigma_0$

- Now the claim looks provable by mathematical induction on $i$
Example of Mathematical Induction (Base Case)

• Base case: \( i = 0 \) or \( \langle W, \sigma_0 \rangle \downarrow \sigma_0 \)
  - To prove an evaluation judgment, construct a derivation tree:

\[
\begin{align*}
\sigma_0(x) &= 6 \\
\langle x, \sigma_0 \rangle &\downarrow 6 \\
\langle 6 \leq 5, \sigma_0 \rangle &\downarrow false \\
\langle x \leq 5, \sigma_0 \rangle &\downarrow false \\
\langle while \ x \leq 5 \ do \ x := x + 1, \sigma_0 \rangle &\downarrow \sigma_0
\end{align*}
\]

• This completes the base case
Example of Mathematical Induction (Inductive Case)

• Must prove $\forall i \in \mathbb{N}. \langle W, \sigma_i \rangle \downarrow \sigma_0 \Rightarrow \langle W, \sigma_{i+1} \rangle \downarrow \sigma_0$
• The beginning of the proof is straightforward
  - Pick an arbitrary $i \in \mathbb{N}$
  - Assume that $\langle W, \sigma_i \rangle \downarrow \sigma_0$
  - Now prove that $\langle W, \sigma_{i+1} \rangle \downarrow \sigma_0$
  - Must construct a derivation tree:

\[
\begin{align*}
&\langle x \leq 5, \sigma_{i+1} \rangle \downarrow \text{true} \\
\hline
&\langle 5 - i, \sigma_{i+1} \rangle \downarrow 5 - i, 5 - i \leq 5 \\
&\hline
&\langle x := x + 1, \sigma_{i+1} \rangle \downarrow \sigma_i \\
&\hline
&\langle W, \sigma_i \rangle \downarrow \sigma_0 \\
&\hline
&\langle x := x + 1; W, \sigma_{i+1} \rangle \downarrow \sigma_0 \\
\end{align*}
\]

\[
\begin{align*}
&\langle x + 1, \sigma_{i+1} \rangle \downarrow 6 - i \\
&\hline
&\langle x := x + 1, \sigma_{i+1} \rangle \downarrow \sigma_i \\
&\hline
&\langle W, \sigma_i \rangle \downarrow \sigma_0 \\
\end{align*}
\]
Discussion

• A proof is more powerful than running the code and observing the result. Why?

• The proof relied on a loop invariant
  - \( x \leq 6 \) in all iterations

• ... and a loop variant
  - \( 6 - x \) is positive and decreasing

• Picking the loop invariant and variant is typically the hardest part of a proof
Discussion

- We proved termination and correctness. This is called **total correctness**

- Mathematical induction is good when we prove properties of natural numbers
  - In PL analysis we most often prove properties of expressions, commands, programs, input data, etc.
  - We need a more powerful induction principle
Well-Founded Induction

• A relation $\prec \subseteq A \times A$ is well-founded if there are no infinite descending chains in $A$
  - Example: $\prec_1 = \{(x, x+1) \mid x \in \mathbb{N}\}$
    • the predecessor relation
  - Example: $\prec = \{(x, y) \mid x, y \in \mathbb{N} \text{ and } x < y\}$

• Well-founded induction:
  - To prove $\forall x \in A. \ P(x)$ it is enough to prove $\forall x \in A. \ (\forall y \prec x \Rightarrow P(y)) \Rightarrow P(x)$

• If $\prec$ is $\prec_1$ then we obtain a special case of mathematical induction

• Why does it work? (see Winskel, Ch 3 for a proof)
Well-Founded Induction. Examples.

- Consider $\prec \subseteq \mathbb{N} \times \mathbb{N}$ with $x \prec y$ iff $x + 2 = y$
  \[ \forall x \in \mathbb{N}. \ (\forall y \prec x \Rightarrow P(y)) \Rightarrow P(x) \text{ is equivalent to} \]
  \[ P(0) \land P(1) \land \forall n \in \mathbb{N}. \ (P(n) \Rightarrow P(n + 2)) \]

- Consider $\prec \subseteq \mathbb{Z} \times \mathbb{Z}$ with $x \prec y$ iff
  \[ (y < 0 \text{ and } y = x - 1) \text{ or } (y > 0 \text{ and } y = x + 1) \]
  
  - $P(0) \land \forall x \leq 0. \ P(x) \Rightarrow P(x - 1) \land \forall x \geq 0. \ P(x) \Rightarrow P(x + 1)$

- 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
  \[ x_2 = x_1 + 1 \lor (x_1 = x_2 \land y_2 = y_1 + 1) \]
  
  - This leads to the induction principle
  \[ P(0,0) \land \forall x,y,y'. \ (P(x,y) \Rightarrow P(x + 1, y') \land P(x, y+1)) \]
  
  - This is sometimes called lexicographic induction
Structural Induction

- Recall $A_{exp}$: $e ::= n \mid e_1 + e_2 \mid e_1 * e_2 \mid x$
- Define $\prec \subseteq A_{exp} \times A_{exp}$ such that
  
  - $e_1 \prec e_1 + e_2$
  - $e_2 \prec e_1 + e_2$
  - $e_1 \prec e_1 * e_2$
  - $e_2 \prec e_1 * e_2$
  - and no other elements of $A_{exp} \times A_{exp}$ are related by $\prec$

- To prove $\forall e \in A_{exp}. P(e)$
  1. Prove $\forall n \in \mathbb{Z}. P(n)$
  2. Prove $\forall x \in \text{Loc}. P(x)$
  3. Prove $\forall e_1, e_2 \in A_{exp}. P(e_1) \land P(e_2) \Rightarrow P(e_1 + e_2)$
  4. Prove $\forall e_1, e_2 \in A_{exp}. P(e_1) \land P(e_2) \Rightarrow P(e_1 * e_2)$
Structural Induction. Notes.

- Called structural induction because the proof is guided by the structure of the expression.

- As many cases as there are expression forms
  - Atomic expressions (with no subexpressions) are all base cases.
  - Composite expressions are the inductive cases.

- This is the most useful form of induction in PL study.
Example of Induction on Structure of Expressions

• Define
  - \( L(e) \): the number of literals and variable occurrences in \( e \)
  - \( O(e) \): the number of operators in \( e \)

• Prove that \( \forall e \in Aexp. \ L(e) = O(e) + 1 \)

• By induction on the structure of \( e \)
  - Case \( e = n \). \( L(e) = 1 \) and \( O(e) = 0 \)
  - Case \( e = x \). \( L(e) = 1 \) and \( O(e) = 0 \)
  - Case \( e = e_1 + e_2 \).
    - \( L(e) = L(e_1) + L(e_2) \) and \( O(e) = O(e_1) + O(e_2) + 1 \)
    - By induction hypothesis \( L(e_1) = O(e_1) + 1 \) and \( L(e_2) = O(e_2) + 1 \)
    - Thus \( L(e) = O(e) + 1 \)
  - Case \( e = e_1 \ast e_2 \). Same as the case for \( + \)
Other Proofs by Structural Induction on Expressions

- Most proofs for Aexp sublanguage of IMP
- Small-step and natural semantics
  \[ \forall e \in \text{Exp. } \forall n \in \mathbb{N}. e \rightarrow^* n \iff e \downarrow n \]
- Natural semantics and denotational semantics
  \[ \forall e \in \text{Exp. } \forall n \in \mathbb{N}. e \downarrow n \iff \llbracket e \rrbracket = n \]
- Small-step and denotational semantics
  \[ \forall e, e' \in \text{Exp. } e \rightarrow e' \Rightarrow \llbracket e \rrbracket = \llbracket e' \rrbracket \]
  \[ \forall e \in \text{Exp. } \forall n \in \mathbb{N}. e \rightarrow^* n \Rightarrow \llbracket e \rrbracket = n \]

- Structural induction on expressions works here because all of the semantics are syntax directed
Another Proof

• Prove that IMP is deterministic

\[\forall e \in Aexp. \forall \sigma \in \Sigma. \forall n, n' \in \mathbb{N}. \langle e, \sigma \rangle \Downarrow n \land \langle e, \sigma \rangle \Downarrow n' \Rightarrow n = n'\]

\[\forall b \in Bexp. \forall \sigma \in \Sigma. \forall t, t' \in \mathbb{B}. \langle b, \sigma \rangle \Downarrow t \land \langle b, \sigma \rangle \Downarrow t' \Rightarrow t = t'\]

\[\forall c \in \text{Com.} \forall \sigma, \sigma', \sigma'' \in \Sigma. \langle c, \sigma \rangle \Downarrow \sigma' \land \langle c, \sigma \rangle \Downarrow \sigma'' \Rightarrow \sigma' = \sigma''\]

• No immediate way to use mathematical induction

• For commands we cannot use induction on the structure of the command

  - Consider the rule for while. Its evaluation does not depend only on the evaluation of its strict subexpressions

\[\langle b, \sigma \rangle \Downarrow \text{true} \quad \langle c, \sigma \rangle \Downarrow \sigma' \quad \langle \text{while } b \text{ do } c, \sigma \rangle \Downarrow \sigma''\]

\[\langle \text{while } b \text{ do } c, \sigma \rangle \Downarrow \sigma''\]
Induction on the Structure of Derivations

- Key idea: The hypothesis does not assume just a $c \in \text{Com}$ 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:

\[
\begin{align*}
<x, \sigma_{i+1}> & \downarrow 5 - i \quad 5 - i \leq 5 \\
\hline
<x \leq 5, \sigma_{i+1}> & \downarrow \text{true} \\
\hline
<x := x+1; W, \sigma_{i+1}> & \downarrow \sigma_0
\end{align*}
\]

\[
\begin{align*}
<x + 1, \sigma_{i+1}> & \downarrow 6 - i \\
\hline
<x := x+1, \sigma_{i+1}> & \downarrow \sigma_i \\
\hline
<W, \sigma_i> & \downarrow \sigma_0
\end{align*}
\]

- Adapt the structural induction principle to work on the structure of derivations

ECS 240 Lecture 3
Induction on Derivations

- To prove that for all derivations $D$ of a judgment, property $P$ holds

1. **For each derivation rule** of the form
   $$\begin{array}{c}
   H_1 \ldots H_n \\
   \hline \\
   C
   \end{array}$$

2. Assume that $P$ holds for derivations of $H_i$ ($i = 1, \ldots, n$)

3. Prove that the property holds for the derivation obtained from the derivations of $H_i$ using the given rule
Example of Induction on Derivations (I)

- Prove that evaluation of commands is deterministic:
  \[ <c, \sigma> \Downarrow \sigma' \Rightarrow \forall \sigma'' \in \Sigma. <c, \sigma> \Downarrow \sigma'' \Rightarrow \sigma' = \sigma'' \]
- Pick arbitrary \( c, \sigma, \sigma' \) and \( D :: <c, \sigma> \Downarrow \sigma' \)
- To prove: \( \forall \sigma'' \in \Sigma. <c, \sigma> \Downarrow \sigma'' \Rightarrow \sigma' = \sigma'' \)
- Proof by induction on the structure of the derivation \( D \)
- Case: last rule used in \( D \) was the one for skip

\[ D :: \quad \text{___________} \]
\[ \langle \text{skip}, \sigma \rangle \Downarrow \sigma \]
- This means that \( c = \text{skip} \), and \( \sigma' = \sigma \)
- By inversion \( <c, \sigma> \Downarrow \sigma'' \) uses the rule for skip. Thus \( \sigma'' = \sigma \)
- This is a base case in the induction
Example of Induction on Derivations (II)

• Case: the last rule used in D was the one for sequencing

\[
D_1 :: \langle c_1, \sigma \rangle \Downarrow \sigma_1 \quad D_2 :: \langle c_2, \sigma_1 \rangle \Downarrow \sigma'
\]

\[
D :: \quad \frac{D_1 :: \langle c_1, \sigma \rangle \Downarrow \sigma_1 \quad D_2 :: \langle c_2, \sigma_1 \rangle \Downarrow \sigma'}{\langle c_1; c_2, \sigma \rangle \Downarrow \sigma'}
\]

• Pick arbitrary \( \sigma'' \) such that \( D'' :: \langle c_1; c_2, \sigma \rangle \Downarrow \sigma'' \).
  - by inversion \( D'' \) uses the rule for sequencing
  - and has subderivations \( D''_1 :: \langle c_1, \sigma \rangle \Downarrow \sigma''_1 \) and \( D''_2 :: \langle c_2, \sigma''_1 \rangle \Downarrow \sigma'' \)

• By induction hypothesis on \( D_1 \) (with \( D''_1 \)): \( \sigma_1 = \sigma''_1 \)
  - Now \( D''_2 :: \langle c_2, \sigma''_1 \rangle \Downarrow \sigma'' \)

• By induction hypothesis on \( D_2 \) (with \( D''_2 \)): \( \sigma'' = \sigma' \)
• This is a simple inductive case
Example of Induction on Derivations (III)

- Case: the last rule used in D was the one for while true

\[ D :: \begin{array}{l}
D_1 :: <b, \sigma> \Downarrow \text{true} \\
D_2 :: <c, \sigma> \Downarrow \sigma_1 \\
D_3 :: <\text{while } b \text{ do } c, \sigma_1> \Downarrow \sigma'
\end{array} \]

- Pick arbitrary \( \sigma'' \) such that \( D'' :: <\text{while } b \text{ do } c, \sigma> \Downarrow \sigma'' \)
  - by inversion and determinism of boolean expressions, \( D'' \) also uses the rule for while true
  - and has subderivations \( D''_2 :: <c, \sigma> \Downarrow \sigma''_1 \) and \( D''_3 :: <W, \sigma''_1> \Downarrow \sigma'' \)
- By induction hypothesis on \( D_2 \) (with \( D''_2 \)): \( \sigma_1 = \sigma''_1 \)
  - Now \( D''_3 :: <\text{while } b \text{ do } c, \sigma_1> \Downarrow \sigma'' \)
- By induction hypothesis on \( D_3 \) (with \( D''_3 \)): \( \sigma'' = \sigma' \)
Induction on Derivation. Notes.

• If we have to prove $\forall x \in A. P(x) \Rightarrow Q(x)$
  - With $x$ 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

• In many situations there are several choices for induction
  - choosing the right one is a trial-and-error process
  - a bit of practice can help a lot
Equivalence

- Two expressions (commands) are equivalent if they yield the same result from all states

\[ e_1 \approx e_2 \iff \forall \sigma \in \Sigma. \forall n \in \mathbb{N}. \langle e_1, \sigma \rangle \downarrow n \iff \langle e_2, \sigma \rangle \downarrow n \]

and for commands

\[ c_1 \approx c_2 \iff \forall \sigma, \sigma' \in \Sigma. \langle c_1, \sigma \rangle \downarrow \sigma' \iff \langle c_2, \sigma \rangle \downarrow \sigma' \]
Notes on Equivalence

• Equivalence is like validity
  - must hold in all states
  - $2 \approx 1 + 1$ is like “$2 = 1 + 1$ is valid”
  - $2 \approx 1 + x$ might or might not hold.
    • So, 2 is not equivalent to $1 + x$

• Equivalence (for IMP) is undecidable
  - If it were we could solve the halting problem. How?

• Equivalence justifies code transformations
  - compiler optimizations
  - code instrumentation
  - abstract modeling

• Semantics is the basis for proving equivalence.
Equivalence Examples

• skip; c ≈ c
• (x := e₁; x := e₂) ≈ x := e₂. When is this true?
• while b do c ≈ if b then c; while b do c else skip
• If e₁ ≈ e₂ then x := e₁ ≈ x := e₂
• while true do skip ≈ while true do x := x + 1
• If c is
  while x ≠ y do
    if x ≥ y then x := x - y else y := y - x
  then (x := 221; y := 527; c) ≈ (x := 17; y := 17)
Proving An Equivalence

• Prove that “skip; c \equiv c” for all c
• Assume that D :: <skip; c, \sigma> \downarrow \sigma'
• By inversion (twice) we have that

\[
D :: \frac{\langle \text{skip, } \sigma \rangle \downarrow \sigma}{\langle \text{skip; } c, \sigma \rangle \downarrow \sigma'} \quad D_1 :: \langle c, \sigma \rangle \downarrow \sigma'
\]

• Thus, we have D_1 :: <c, \sigma> \downarrow \sigma'
• The other direction is similar
Proving An Inequivalence

• Prove that \( x := y \not\equiv x := z \) when \( y \neq z \)
• It suffices to exhibit a state \( \sigma \) in which the two commands yield different results

• Let \( \sigma(y) = 0 \) and \( \sigma(z) = 1 \)
• Then \( \langle x := y, \sigma \rangle \downarrow \sigma[x := 0] \)
• and \( \langle x := z, \sigma \rangle \downarrow \sigma[x := 1] \)
Summary of Operational Semantics

• Precise specification of dynamic semantics
  - order of evaluation (or that it doesn’t matter)
  - error conditions (sometimes implicitly, by rule applicability)

• Simple and abstract (vs. implementations)
  - no low-level details such as stack and memory management, data layout, etc.

• Often not compositional (see while)

• Basis for some proofs about the language

• Basis for some reasoning about programs

• Point of reference for other semantics