Contextual Semantics
- Contextual semantics is a small-step semantics where the atomic execution step is a rewrite of the program.
- We will define a relation $<c, σ> → <c', σ'>$.
  - $c'$ is obtained from $c$ through an atomic rewrite step.
  - E.g.:
    
    $<x := 1+(2+8), σ> → <x := 1+10, σ> → <x := 1+10, σ> → <skip, σ[x := 10]>$
    
    - Evaluation terminates when the program has been rewritten to a terminal program.
    - For IMP the terminal command is "skip".
    - As long as the command is not "skip" we can make progress.
    - Some commands never reduce to skip (e.g., while true do skip).

Redexes
- A redex is a syntactic expression or command that can be reduced (transformed) in one atomic step.
- For brevity, we mix expression and command redexes (and also omit some redexes and contexts).
- Redexes are defined by a grammar:
  
  $r ::= x | n_1 + n_2 | x := n | skip; c | if true then c_1 else c_2 | if false then c_1 else c_2 | while b do c$

  - Note that $(1 * 3) + 2$ is not a redex, but $1 + 3$ is.

Local Reduction Rules for IMP
- One for each redex: $<r, σ> → <e, σ'>$.
  - This means that in state $σ$, the redex $r$ can be replaced in one step with the expression $e$.
  - E.g.:
    
    $<x, σ> → <σ(x), σ>$
    $<n_1 + n_2, σ> → <n, σ>$ where $n = n_1 + n_2$
    $<n_1 = n_2, σ> → <true, σ>$ if $n_1 = n_2$
    $<x := n, σ> → <skip, σ[x := n]>$
    $<skip; c, σ> → <c, σ>$
    $<if true then c_1 else c_2, σ> → <c_1, σ>$
    $<if false then c_1 else c_2, σ> → <c_2, σ>$
    $<while b do c, σ> → if b then c; while b do c else skip, σ>$

Review
- A redex is something that can be reduced in one step.
  - E.g. 2+8
- Local reduction rules reduce these redexes.
  - E.g. $<2+8, σ> → <10, σ>$

Next: global reduction rules
- Consider:
  - $<x := 1+(2+8), σ>$
  - $<while false do x := 1+(2+8), σ>$
- Should we also reduce 2+8 in these cases?

Contexts
- A context is an expression or command with exactly one marker "*".
  - The marker is sometimes called a hole.
  - $H[e]$ is obtained from $H$ by replacing the marker "*" with $e$

- Examples
  - $x := 1*$
    
    - Fill context $H$ with 2+8 to yield $H[2+8] = x := 1+(2+8)$
    - Or fill context with 10 to yield $H[10] = x := 1+10$
    - While false do $x := 1*$
      
      - Fill with 2+8 to yield $H[2+8] = while false do x := 1+(2+8)$
      - While false do $*$
Evaluation Contexts

- An evaluation context is a context in which the marker indicates the next place for evaluation.
  - identifies the next redex, a bit like a program counter

\[
H ::= \\
| H + e \\
| n + H \\
| x := H \\
| \text{if } H \text{ then } c_1 \text{ else } c_2 \\
| H; c
\]

Examples
- \(x := 1+\)
- \(\)
- NOT: while false do \(x := 1+\)
- NOT: if \(b\) then \(c\) else \(\)

Contexts: Notes

- Evaluation contexts say how to find the next redex:
  - Consider \(e_1 + e_2\) and its decomposition as \(H[r]\).
  - If \(e_1 = n_1\) and \(e_2 = n_2\)
    - then \(H = n_1 + n_2\).
  - If \(e_1 = n_1\) and \(e_2 \neq n_2\)
    - then \(H = n_1 + H_2\) and \(e_2 = H_2[r]\).
  - If \(e_1 \neq n_1\)
    - then \(H = H_1 + H_2\) and \(e_1 = H_1[r]\).
  - In the last two cases the decomposition is done recursively.
  - In each case the solution is unique.

The Global Reduction Rule

- General idea of the contextual semantics:
  - Decompose the current expression into
    - the next redex \(r\)
    - an evaluation context \(H\) (the remaining program)
  - Reduce the redex "\(r\)" to some other expression "\(e\)."
  - Put "\(e\)" back into the original context, yielding \(H[e]\).

- Formalized as a small step rule:

\[
\text{If } <r, \sigma> \rightarrow <e, \sigma'> \text{ then } <H[r], \sigma> \rightarrow <H[e], \sigma'>
\]

The Global Reduction Rule: Example

- Consider the command \(x := 1+(2+8)\)
- Split into an evaluation context \(H\) and a redex \(r\)
- Get

\[
H = x := 1+ \\
H[r] = x := 1+(2+8) \quad \text{(original command)}
\]
- Have

\[
2+8, \sigma \rightarrow 10, \sigma \quad \text{(local reduction rule)}
\]
- Define global reduction

\[
2+8, \sigma \rightarrow 10, \sigma \quad \text{or, equivalently}
\]

\[
x := 1+(2+8), \sigma \rightarrow x := 1+10, \sigma
\]

Contextual Semantics: Example

- Consider the small-step evaluation of \(x := 1; x := x + 1\) in the initial state \([x := 0]\)

<table>
<thead>
<tr>
<th>State</th>
<th>Context</th>
<th>Redex</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x = 1; x := x + 1, [x := 0])</td>
<td>(x, x := x + 1)</td>
<td>(x := 1)</td>
</tr>
<tr>
<td>skip; (x := x + 1, [x := 1])</td>
<td>()</td>
<td>(\text{skip; } x := x + 1)</td>
</tr>
<tr>
<td>(x := x + 1, [x := 1])</td>
<td>(x := * + 1)</td>
<td>(x := x + 1)</td>
</tr>
<tr>
<td>(x := 1 + 1, [x := 1])</td>
<td>(x := *)</td>
<td>(1 + 1)</td>
</tr>
<tr>
<td>(x := 2, [x := 1])</td>
<td>()</td>
<td>(x := 2)</td>
</tr>
<tr>
<td>skip; (x := 2)</td>
<td>()</td>
<td>()</td>
</tr>
</tbody>
</table>
Normal vs Short-Circuit Boolean Operators

- What if we want normal evaluation of \( \land \)?
  - Define the following contexts, redexes, and local rules:
    \[
    H ::= \ldots | H \land b_2 | p_1 \land H \\
    r ::= \ldots | p_1 \land p_2 \\
    \langle p_1 \land p_2, \sigma \rangle \rightarrow \langle p, \sigma \rangle \quad \text{where } p = p_1 \land p_2
    \]

- What if we want short-circuit evaluation of \( \land \)?
  - Define the following contexts, redexes, and local rules:
    \[
    H ::= \ldots | H \land b_2 \\
    r ::= \ldots | \text{true} \land b_2 | \text{false} \land b_2 \\
    \langle \text{true} \land b_2, \sigma \rangle \rightarrow \langle b_2, \sigma \rangle \\
    \langle \text{false} \land b_2, \sigma \rangle \rightarrow \langle \text{false}, \sigma \rangle
    \]
  - The local reduction kicks in before \( b_2 \) is evaluated.

Contextual Semantics: Notes

- One can think of the \( \bullet \) as representing the program counter.
- The advancement rules for \( \bullet \) are not trivial:
  - At each step the entire command is decomposed.
  - This makes contextual semantics inefficient to implement directly.
- The major advantage of contextual semantics is that it 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.

Some Further Topics

- Treatment of errors in operational semantics
  - with an explicit "error" result, as in \((3/0) \rightarrow \text{error}\)
  - with an "error" expression, as in \((3 + \text{error})\)
  - with "stuck" computations, so \((3/0) \rightarrow r\) for no \( r \)
- Treatment of overflow (see homework 2)

Guidelines for the Final Project

- For example: \( c = c_1; c_2 \)
  - either \( c_1 = \text{skip} \) and then \( c = H[\text{skip}; c_2] \) with \( H = \bullet \)
  - or \( c_1 = \text{skip} \) and then \( c_1 = H[r] \):
    \[ c = c_1; c_2 = H[r]; c_2 = H[r] \text{ where } H = H; c_2 \]
- For example: \( c = \text{if } b \text{ then } c_1 \text{ else } c_2 \)
  - either \( b = \text{true} \) or \( b = \text{false} \) and then \( c = H[r] \) with \( H = \bullet \)
  - or \( b \) is not a value and \( b = H[r] \):
    \[ c = H[r] \text{ where } H = \text{if } H \text{ then } c_1 \text{ else } c_2 \]
- Decomposition theorem: If \( c \) is not "skip" then there exist unique \( H \) and \( r \) such that \( c = H[r] \).
  \( \Rightarrow \) Progress and determinism.
The Final Project

- Three kinds
  - small survey of recent work on a relevant topic (individual)
  - programming project
  - research paper

- Team (1-4 people) or individual projects

- Scale
  - 20-40+ hours of work per person
  - short report
  - short presentation

Picking a Project

- You are encouraged to define your own project.

- If you prefer it, I will be happy to assign you a project, but I can’t guarantee that you will be happy with it.

Scale

- I don’t expect very fancy projects.
  - 20-40 hours of work should suffice, unless you are very enthusiastic.

- However, you are welcome to tackle much more ambitious projects.
  - You should structure such a project so that you can show partial results this quarter.
  - Staged development very often a good idea

Collaboration

- You can do the projects (except for surveys) in groups of 1-4.

- If you have a great project idea, and it looks too big for one person, feel free to recruit help.

- I think that it is easier to do the projects alone, but you are welcome to make your own choices.

Cheating

- Projects should be new and original
  - not a cut-and-paste of prior work,
  - not also fulfilling the requirements of another course
  - not something you have already finished.

- But it is good if you care about the project beyond completion of this course.

- In a group project, you are expected to do your share.
  - You should notify me if others are not doing theirs.

Kinds of Projects

- There are three basic kinds of projects:
  - Survey
    - of work in some area of programming languages.
  - Implementation
    - of a small language or algorithm.
  - Research
    - on programming languages.

- These kinds can be combined to some extent.

- In all cases, you must write a 2-10 page report.
The Survey Project

- Pick an area in which you are interested.
- For example:
  - a family of domain-specific languages,
  - prototyping environments for language design,
  - integration of static and dynamic scoping,
  - integration of static and dynamic typing,
  - implementation strategies for polymorphism,
  - concurrency primitives and objects,
  - type inference for object-oriented languages,
  - axiomatic semantics for parallel languages.
- For more ideas, see for example the proceedings of recent POPL, PLDI, or OOPSLA conferences.

The Survey Project (cont.)

- Read well 2-4 papers.
- Read at least superficially 2-4 extra papers.
- Write a short report on what you have learned.
  - What are the basic problems in this area?
  - What are the basic approaches to solving them?
  - What are the main achievements to date?

The Implementation Project

- Implement a small language or an algorithm related to language design, e.g.:
  - an interesting type checking algorithm,
  - a type inference algorithm,
  - some other static analysis algorithm, perhaps based on axiomatic semantics,
  - a big/small step operational semantics,
  - a real language that you invented in the past.
- Write a short report on your project (1-2 pages).
- How much code?
  - 100 lines is probably too small; 10,000 is probably too big.
  - Writing a program in an interesting language is not in itself sufficient!

The Research Project

- Research projects are the hardest.
- There are several sorts of research projects:
  - Design: invent a language, or part of a language.
  - Modelling: try to formalize some interesting aspect of some existing languages.
  - Theory: extend the theory of language design.
  - Implementation - as discussed earlier.
- In all cases, write a report on this work, of whatever length is appropriate.
  - The writing need not be publication-quality, but I should be able to read it easily.

Reports and Presentations

- Project reports are due at start of last class on December 2nd.
- We will have brief presentations in class during the last week of classes.

Some Proof Techniques for Language Analysis
Plan

- We will study various flavors of induction.

Induction

- Probably the single most important technique for the study of formal semantics and type systems of programming languages.
- Of several kinds
  - mathematical induction (the simplest)
  - well-founded induction (the most general)
  - structural induction (the most widely used in this context)

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 step looks similar to the goal but it is simpler because of the assumption that \( P(n) \) holds.
  \( \forall n \in \mathbb{N}. P(n-1) \Rightarrow P(n) \) vs. \( \forall n \in \mathbb{N}. P(n) \)
- Why does mathematical induction work?
  - The key property of \( \mathbb{N} \) is that there are no infinite descending chains of naturals.
  - 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 \( a(x) \leq 6 \) then
  \( \langle \text{while } x \leq 5 \text{ do } x := x + 1, \sigma \rangle \Rightarrow \sigma \)
- Reformulate the claim:
  - Let \( W = \text{while } x \leq 5 \text{ do } x := x + 1 \)
  - Let \( a_i = a(x) = 6 \)
  - Claim: \( \forall i \in \mathbb{N}. \langle W, a_i \rangle \Rightarrow \sigma_0 \)
- Now the claim looks provable by mathematical induction on \( i \).

Example of Mathematical Induction (Base Case)

- Base case: \( i = 0 \) or \( \langle W, a_0 \rangle \uparrow \sigma_0 \)
  - To prove an evaluation judgment, construct a derivation tree:
    \[ \sigma_0(x) = 6 \]
    \[ \langle x, a_0 \rangle \uparrow 6 \]
    \[ \langle 5, a_0 \rangle \uparrow 5 \]
    \[ \langle x \leq 5, a_0 \rangle \uparrow \text{false} \]
    \[ \langle \text{while } x \leq 5 \text{ do } x := x + 1, \sigma_0 \rangle \]
- This completes the base case.
**Example of Mathematical Induction (Inductive Case)**

- We 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 $\langle W, \sigma_{i+1} \rangle \Downarrow \sigma_0$
  - We must construct a derivation tree:

- Example:
  - $\langle x, \sigma_i \rangle \Downarrow 5 - i \quad 5 - i \leq 5$
  - $\langle x, \sigma_i \rangle \Downarrow \text{true}$
  - $\langle x \leftarrow x + 1; W, \sigma_i \rangle \Downarrow \sigma_0$
  - $\langle \text{while } x \leq 5 \text{ do } x := x + 1, \sigma_i \rangle \Downarrow \sigma_0$
  - $\langle W, \sigma_i \rangle \Downarrow \sigma_0$

**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.

**Well-Founded Induction**

- A relation $<_1 \subseteq A \times A$ is well-founded if there are no infinite descending chains in $A$.
  - Example: $<_1 = \{(x, x+1) \mid x \in \mathbb{N}\}$
  - the predecessor relation
- Example: $<_2 = \{(x, y) \mid x, y \in \mathbb{N} \text{ and } x < y\}$
- Well-founded induction:
  - To prove $\forall x \in A. P(x)$ it is enough to prove $\forall x \in A. \left( \forall y. P(y) \Rightarrow P(x) \right)$
- If $<_1$ is $<_1$, then we obtain a special case of mathematical induction.

**Well-Founded Induction: Examples**

- Consider $<_1 \subseteq \mathbb{N} \times \mathbb{N}$ with $x <_1 y$ iff $x + 2 > y$
  - $\forall x \in \mathbb{N}. (\forall y < x \Rightarrow P(y)) \Rightarrow P(x)$ is equivalent to $P(0) \land (\forall n \in \mathbb{N}. P(n) \Rightarrow P(n + 2))$
- Consider $<_2 \subseteq \mathbb{Z} \times \mathbb{Z}$ with $x <_2 y$ iff $(y < 0 \land y \leq x - 1)$ or $(y > 0 \land y \geq x + 1)$
  - $\forall x \in \mathbb{Z}. (\forall y < x \Rightarrow P(y)) \Rightarrow P(x)$ is equivalent to $P(0) \land (\forall x \leq 0. P(x) \Rightarrow P(x - 1)) \land (\forall x > 0. P(x) \Rightarrow P(x + 1))$
Well-Founded Induction: Examples (cont.)

- Consider \( p \subseteq (\mathbb{N} \times \mathbb{N}) \times (\mathbb{N} \times \mathbb{N}) \) 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
    \[
    \begin{align*}
    & P(0,0) \\
    & \quad \land \forall x, y, y'. (P(x,y) \implies P(x + 1, y'))
    \end{align*}
    \]
  - This is sometimes called lexicographic induction.

Structural Induction

- Recall \( e ::= n | e_1 + e_2 | e_1 \cdot e_2 | x \)
- Define \( p \subseteq A_{exp} \times A_{exp} \) such that
  \[
  \begin{align*}
  & e_1 \prec e_1 + e_2 \\
  & e_2 \prec e_1 + e_2 \\
  & e_1 \prec e_1 \cdot e_2 \\
  & e_2 \prec e_1 \cdot e_2
  \end{align*}
  \]
  - 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 \( \forall x \in L. P(x) \)
  3. Prove \( \forall e_1, e_2 \in A_{exp}. P(e_1) \land P(e_2) \implies P(e_1 + e_2) \)
  4. Prove \( \forall e_1, e_2 \in A_{exp}. P(e_1) \land P(e_2) \implies P(e_1 \cdot e_2) \)

Structural Induction: Notes

- It is called structural induction because proofs are guided by the structure of the expression.
- In a proof, there are 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 the study of programming languages.

Example of Induction on Structure of Expressions

- Let
  \[
  \begin{align*}
  & L(e) \text{ be the number of literals and variable occurrences in } e \\
  & O(e) \text{ be the number of operators in } e
  \end{align*}
  \]
- Prove that \( \forall e \in A_{exp}. L(e) = O(e) + 1 \)
- 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 \( e = e_1 + e_2 \).
    \[
    \begin{align*}
    & L(e) = L(e_1) + L(e_2) \\
    & O(e) = O(e_1) + O(e_2) + 1
    \end{align*}
    \]
    - 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 \( e = e_1 \cdot e_2 \) same as the case for \( + \).

Other Proofs by Structural Induction on Expressions

- Most proofs for ARITH (or the Aexp and Bexp sublanguages of IMP).
- Small-step vs. natural semantics
  \[
  \begin{align*}
  & \forall e \in Exp. \forall n \in \mathbb{N}. n \rightarrow e \quad \Rightarrow \\
  & \forall e \in Bexp. \forall n \in \mathbb{N}. n \rightarrow e \quad \Rightarrow \\
  & \forall c \in Comm. \forall n, d'. n \rightarrow d' \quad \Rightarrow
  \end{align*}
  \]
- Structural induction on expressions works here because all of the semantics are syntax directed.

Another Proof

- Prove that IMP is deterministic
  \[
  \begin{align*}
  & \forall e \in A_{exp}. \forall \sigma \in \Sigma. \forall n, \sigma' \in A_{exp}. \sigma_0 \cup \sigma \land \sigma_1 \cup \sigma \lor \sigma_0 \cup \sigma' \lor \sigma_1 \cup \sigma' \quad \Rightarrow \\
  & \forall d, d' \in A_{exp}. \forall n, \sigma, \sigma'. \sigma_0 \cup \sigma \land \sigma_1 \cup \sigma \lor \sigma_0 \cup \sigma' \lor \sigma_1 \cup \sigma' \quad \Rightarrow
  \end{align*}
  \]
- 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
    \[
    \begin{align*}
    & <b, \sigma > \downarrow \text{ true} \\
    & <c, \sigma > \uparrow \sigma' \\
    & \langle \text{while } b \text{ do } c, \sigma \rangle \uparrow \sigma''
    \end{align*}
    \]
  - Prove that \( \langle b, \sigma \rangle \downarrow \text{ true} \)
Induction on the Structure of Derivations

- Key idea: The hypothesis gives not only a $c \in \text{Comm}$ but also the existence of a derivation of $<c, \sigma_0> \Downarrow \sigma'$.
- Derivation trees are also defined inductively, just like expression trees.
- A derivation is built of subderivations:
  - We adapt the structural induction principle to work on the structure of derivations.

\[
\begin{align*}
\text{while } x \leq 5 \text{ do } x := x + 1, \sigma_i+1 \Downarrow \sigma_0 \\
<\text{while } x \leq 5 \text{ do } x := x + 1, \sigma_i+1> \Downarrow \sigma_0
\end{align*}
\]

Example of Induction on Derivations (I)

- Proof 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 $D :: <c, \sigma> \Downarrow \sigma''$.

\[
\begin{align*}
\text{Case: the last rule used in } D \text{ was the one for skip:} \\
\text{D :: } <\text{skip}, \sigma> \Downarrow \sigma'
\end{align*}
\]

- By inversion $D$ 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 while true

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

- Pick arbitrary $\sigma'$ such that $D'' :: <\text{while } b \text{ do } c, \sigma_1> \Downarrow \sigma''$.
- By induction hypothesis on $D_2$ (with $D''_3$): $\sigma_1'' = \sigma''$.
- 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

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

- Pick arbitrary $\sigma''$ such that $D'' :: <\text{while } b \text{ do } c, \sigma_1> \Downarrow \sigma''$.
- By induction hypothesis on $D_3$ (with $D''_3$): $\sigma_1'' = \sigma''$.
- By induction hypothesis on $D_2$ (with $D''_2$): $\sigma'' = \sigma$.

Induction on Derivations

- To prove that for all derivation $D$ of a judgment, property $P$ holds:
  1. For each derivation rule of the form $H_1 \ldots H_n$ to $C$
  2. Assume that $P$ holds for a derivation of $H_i$ ($i = 1, \ldots, n$).
  3. Prove the property holds for the derivation obtained from the derivations of $H_i$ using the given rule.

Induction on Derivation: Notes

- If we have to prove $\forall x \in A. P(x) \Rightarrow Q(x)$
- With $A$ inductively defined and $P(x)$ rule-defined
- If $x \in A$ 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 safer.
- In many situations there are several choices for induction.
- Choosing the right one is a trial-and-error process.
- A lot of practice can help a lot!
Equivalence

- Two expressions (commands) are equivalent if they yield the same result from all states
  
  \( e_1 = e_2 \iff \forall \sigma \in \Sigma. \forall n \in \mathbb{Z}. <e_1, \sigma> \Downarrow n \iff <e_2, \sigma> \Downarrow n \)

and for commands

\( c_1 = c_2 \iff \forall \sigma, \sigma' \in \Sigma. <c_1, \sigma> \Downarrow \sigma' \iff <c_2, \sigma> \Downarrow \sigma' \)

Notes on Equivalence

- Equivalence is like validity:
  - It must hold in all states.
  - \( 2 = 1 + 1 \) is like "2 = 1 + 1 is valid".
  - \( 2 = 1 + x \) might or might not hold.
    - So, 2 is not equivalent to \( 1 + x \)
- Equivalence (for IMP) is undecidable.
  - If it were decidable 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
- \((x := e_1; x := e_2) = x := e_2\). (When is this true?)
- while b do c = if b then c; while b do c else skip
- If \( e_1 = e_3 \) then \( x := e_1 = x := e_3 \)
- while true do skip = while true do \( x := x + 1 \)
- If \( c \) is
  - while \( x = y \) do
    - if \( x \geq y \) then \( x := x - y \); \( y := y - x \)
  - then \( (x := 221; y := 527; c) = (x := 17; y := 17) \)

Proving an Equivalence

- Prove that "skip; c = c" for all c.
- Assume that \( D :: <\text{skip; } c, \sigma> \Downarrow \sigma' \).
  - By inversion (twice) we have that:
    \[
    D :: <\text{skip; } c, \sigma> \Downarrow \sigma' \\
    D_1 :: <c, \sigma> \Downarrow \sigma'
    \]
  - Thus, we have \( D_1 :: <c, \sigma> \Downarrow \sigma' \).
  - The other direction is similar.

Proving an Inequivalence

- Prove that \( x := y \neq x := z \) when \( y \neq z \).
- It suffices to exhibit a \( \sigma \) in which the two commands yield different results,
  - Let \( \sigma(y) = 0 \) and \( \sigma(z) = 1 \).
  - Then \( \sigma(x) = 0 \), \( \sigma(x) = 1 \)
  - and \( \sigma(x) = 0 \) and \( \sigma(x) = 1 \).

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 (cf. implementations)
  - no low-level details such as stack and memory management, data layout, etc.
- Often not compositional (as for while)
- Basis for some proofs about languages
- Basis for some reasoning about particular programs
- Point of reference for other semantics