

Computer Science 203  
 Programming Languages  
 Fall 2004 - Lecture 4

Cormac Flanagan

University of California, Santa Cruz

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  $\langle c, \sigma \rangle \rightarrow \langle c', \sigma' \rangle$ 
  - $c'$  is obtained from  $c$  through an atomic rewrite step.
  - E.g.:  $\langle x := 1+(2+8), \sigma \rangle$ 
    - $\rightarrow \langle x := 1+10, \sigma \rangle$
    - $\rightarrow \langle x := 1+10, \sigma \rangle$
    - $\rightarrow \langle \text{skip}, \sigma[x:=10] \rangle$
  - 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
      | n1 + n2
      | x := n
      | skip; c
      | if true then c1 else c2
      | if false then c1 else c2
      | 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:  $\langle r, \sigma \rangle \rightarrow \langle e, \sigma' \rangle$ 
    - This means that in state  $\sigma$ , the redex  $r$  can be replaced in one step with the expression  $e$ .
- ```

<x, σ>      → <σ(x), σ>
<n1 + n2, σ> → <n, σ>           where n = n1 + n2
<n1 = n2, σ> → <true, σ>         if n1 = n2
<x := n, σ> → <skip, σ[x := n]>
<skip; c, σ> → <c, σ>
<if true then c1 else c2, σ> → <c1, σ>
<if false then c1 else c2, σ> → <c2, 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.  $\langle 2+8, \sigma \rangle \rightarrow \langle 10, \sigma \rangle$
- Next: **global reduction rules**
- Consider
  - $\langle x := 1+(2+8), \sigma \rangle$
  - $\langle \text{while false do } x := 1+(2+8), \sigma \rangle$
- Should we also reduce  $2+8$  in these cases?

Contexts

- A **context** is an expression or command with exactly one marker " $\bullet$ "
  - The marker is sometimes called a hole.
  - $H[e]$  is obtained from  $H$  by replacing the marker  $\bullet$  with  $e$
- Examples
  - $x := 1+\bullet$ 
    - Fill context  $H$  with  $2+8$  to yield  $H[2+8] = x := 1+(2+8)$
    - Or fill context with  $10$  to yield  $H[10] = x := 1+10$
  - $\text{while false do } x := 1+\bullet$ 
    - Fill with  $2+8$  to yield  $H[2+8] = \text{while false do } x := 1+(2+8)$
  - $\bullet$

## 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
      | if H then c1 else c2
      | H; c
```

CMPS201 Lecture 4

7

## 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
      | if H then c1 else c2
      | H; c
```

- Examples

- $x := 1+•$
- $•$
- NOT: while false do  $x := 1+•$
- NOT: if b then c else  $•$

CMPS201 Lecture 4

8

## Contexts: Notes

- Evaluation contexts say how to find the next redex:
  - Consider  $e_1 + e_2$  and its decomposition as  $H[r]$ .
  - If  $e_1$  is  $n_1$  and  $e_2$  is  $n_2$ 
    - then  $H = •$  and  $r = n_1 + n_2$ .
  - If  $e_1$  is  $n_1$  and  $e_2$  is not  $n_2$ 
    - then  $H = n_1 + H_2$  and  $e_2 = H_2[r]$ .
  - If  $e_1$  is not  $n_1$ 
    - then  $H = H_1 + e_2$  and  $e_1 = H_1[r]$ .
  - In the last two cases the decomposition is done recursively.
  - In each case the solution is unique.

CMPS201 Lecture 4

9

## The Global Reduction Rule

- General idea of the contextual semantics:
  - Decompose the current expression into
    - the next redex  $r$
    - and 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:

If  $\langle r, \sigma \rangle \rightarrow \langle e, \sigma' \rangle$  then  $\langle H[r], \sigma \rangle \rightarrow \langle H[e], \sigma' \rangle$

CMPS201 Lecture 4

10

## 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+•
r = 2+8
H[r] = x := 1+(2+8) (original command)
```
- Have
  - $\langle 2+8, \sigma \rangle \rightarrow \langle 10, \sigma \rangle$  (local reduction rule)
- Define global reduction
  - $\langle H[2+8], \sigma \rangle \rightarrow \langle H[10], \sigma \rangle$  or, equivalently
  - $\langle x := 1+(2+8), \sigma \rangle \rightarrow \langle x := 1+10, \sigma \rangle$

CMPS201 Lecture 4

11

## Contextual Semantics: Example

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

| State                                               | Context         | Redex                     |
|-----------------------------------------------------|-----------------|---------------------------|
| $\langle x := 1; x := x + 1, [x := 0] \rangle$      | $•; x := x + 1$ | $x := 1$                  |
| $\langle \text{skip}; x := x + 1, [x := 1] \rangle$ | $•$             | $\text{skip}; x := x + 1$ |
| $\langle x := x + 1, [x := 1] \rangle$              | $x := • + 1$    | $x$                       |
| $\langle x := 1 + 1, [x := 1] \rangle$              | $x := •$        | $1 + 1$                   |
| $\langle x := 2, [x := 1] \rangle$                  | $•$             | $x := 2$                  |
| $\langle \text{skip}, [x := 2] \rangle$             |                 |                           |

CMPS201 Lecture 4

12

## Normal vs Short-Circuit Boolean Operators

- What if we want normal evaluation of  $\wedge$  ?
  - Define the following contexts, redexes, and local rules:

$H ::= \dots \mid H \wedge b_2 \mid p_1 \wedge H$

$r ::= \dots \mid p_1 \wedge p_2$

$\langle p_1 \wedge p_2, \sigma \rangle \rightarrow \langle p, \sigma \rangle$  where  $p = p_1 \wedge p_2$

## Normal vs Short-Circuit Boolean Operators

- What if we want normal evaluation of  $\wedge$  ?
  - Define the following contexts, redexes, and local rules:

$H ::= \dots \mid H \wedge b_2 \mid p_1 \wedge H$

$r ::= \dots \mid p_1 \wedge p_2$

$\langle p_1 \wedge p_2, \sigma \rangle \rightarrow \langle p, \sigma \rangle$  where  $p = p_1 \wedge p_2$

- What if we want short-circuit evaluation of  $\wedge$  ?

- Define the following contexts, redexes, and local rules:

$H ::= \dots \mid H \wedge b_2$

$r ::= \dots \mid \text{true} \wedge b_2 \mid \text{false} \wedge b_2$

$\langle \text{true} \wedge b_2, \sigma \rangle \rightarrow \langle b_2, \sigma \rangle$

$\langle \text{false} \wedge b_2, \sigma \rangle \rightarrow \langle \text{false}, \sigma \rangle$

- 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)

## Contextual Semantics: Notes

- For example:  $c = c_1; c_2$ 
  - either  $c_1 = \text{skip}$  and then  $c = H[\text{skip}; c_2]$  with  $H = \bullet$
  - or  $c_1 \neq \text{skip}$  and then  $c_1 = H[r]$ ; so  $c = c_1; c_2 = H[r]; c_2 = H'[r]$  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  $b = H[r]$ ; so  $c = H'[r]$  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  $c$  is  $H[r]$ .  
 $\Rightarrow$  Progress and determinism.

## Guidelines for the Final Project

## 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 2<sup>nd</sup>**.
- 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  $\sigma(x) \leq 6$  then  
 $\langle \text{while } x \leq 5 \text{ do } x := x + 1, \sigma \rangle \Downarrow \sigma[x := 6]$
- Reformulate the claim:
  - Let  $W = \text{while } x \leq 5 \text{ do } x := x + 1$
  - Let  $\sigma_i = \sigma[x := 6 - i]$
  - Claim:  $\forall i \in \mathbb{N}. \langle W, \sigma_i \rangle \Downarrow \sigma_0$
- 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:

$$\frac{\sigma_0(x) = 6}{\langle x, \sigma_0 \rangle \Downarrow 6} \quad \langle 5, \sigma_0 \rangle \Downarrow 5$$
$$\frac{\langle x \leq 5, \sigma_0 \rangle \Downarrow \text{false}}{\langle \text{while } x \leq 5 \text{ do } x := x + 1, \sigma_0 \rangle \Downarrow \sigma_0}$$

- This completes the base case.



### Well-Founded Induction: Examples (cont.)

- Consider  $< \subseteq (\mathbb{N} \times \mathbb{N}) \times (\mathbb{N} \times \mathbb{N})$  and  $(x_1, y_1) < (x_2, y_2)$  iff  $x_2 = x_1 + 1 \vee (x_1 = x_2 \wedge y_2 = y_1 + 1)$ 
  - This leads to the induction principle  $P(0,0) \wedge \forall x,y,y'. (P(x,y) \Rightarrow P(x+1,y) \wedge P(x,y+1))$
  - This is sometimes called lexicographic induction.

### Structural Induction

- Recall  $e ::= n \mid e_1 + e_2 \mid e_1 * e_2 \mid x$
- Define  $< \subseteq \text{Aexp} * \text{Aexp}$  such that
  - $e_1 < e_1 + e_2$
  - $e_2 < e_1 + e_2$
  - $e_1 < e_1 * e_2$
  - $e_2 < e_1 * e_2$
- and no other elements of  $\text{Aexp} * \text{Aexp}$  are related by  $<$
- To prove  $\forall e \in \text{Aexp}. P(e)$ 
  - Prove  $\forall n \in \mathbb{Z}. P(n)$
  - Prove  $\forall x \in L. P(x)$
  - Prove  $\forall e_1, e_2 \in \text{Aexp}. P(e_1) \wedge P(e_2) \Rightarrow P(e_1 + e_2)$
  - Prove  $\forall e_1, e_2 \in \text{Aexp}. P(e_1) \wedge P(e_2) \Rightarrow P(e_1 * e_2)$

### 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
  - $L(e)$  be the number of literals and variable occurrences in  $e$
  - $O(e)$  be the number of operators in  $e$
- Prove that  $\forall e \in \text{Aexp}. L(e) = O(e) + 1$
- 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  $e = e_1 * e_2$ : 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
  - $\forall e \in \text{Exp}. \forall n \in \mathbb{Z}. e \rightarrow^* n \Leftrightarrow e \Downarrow n$
- Structural induction on expressions works here because all of the semantics are syntax directed.

### Another Proof

- Prove that IMP is deterministic
    - $\forall e \in \text{Aexp}. \forall \sigma \in \Sigma. \forall n, n' \in \mathbb{N}. \langle e, \sigma \rangle \Downarrow n \wedge \langle e, \sigma \rangle \Downarrow n' \Rightarrow n = n'$
    - $\forall b \in \text{Bexp}. \forall \sigma \in \Sigma. \forall t, t' \in \mathbb{B}. \langle b, \sigma \rangle \Downarrow t \wedge \langle b, \sigma \rangle \Downarrow t' \Rightarrow t = t'$
    - $\forall c \in \text{Comm}. \forall \sigma, \sigma', \sigma'' \in \Sigma. \langle c, \sigma \rangle \Downarrow \sigma' \wedge \langle c, \sigma \rangle \Downarrow \sigma'' \Rightarrow \sigma' = \sigma''$
  - 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
- $$\frac{\langle b, \sigma \rangle \Downarrow \text{true} \quad \langle c, \sigma \rangle \Downarrow \sigma' \quad \langle \text{while } b \text{ do } c, \sigma \rangle \Downarrow \sigma''}{\langle \text{while } b \text{ do } c, \sigma \rangle \Downarrow \sigma''}$$

### Induction on the Structure of Derivations

- Key idea: The hypothesis gives not only a  $c \in \text{Comm}$  but also the existence of a derivation of  $\langle c, \sigma \rangle \Downarrow \sigma'$ .
- Derivation trees are also defined inductively, just like expression trees.
- A derivation is built of subderivations:

$$\frac{\frac{\langle x, \sigma_{i+1} \rangle \Downarrow 5-i \quad 5-i \leq 5}{\langle x \leq 5, \sigma_{i+1} \rangle \Downarrow \text{true}} \quad \frac{\frac{\langle x+1, \sigma_{i+1} \rangle \Downarrow 6-i}{\langle x := x+1, \sigma_{i+1} \rangle \Downarrow \sigma_i} \quad \langle W, \sigma_i \rangle \Downarrow \sigma_0}{\langle x := x+1; W, \sigma_{i+1} \rangle \Downarrow \sigma_0}}{\langle \text{while } x \leq 5 \text{ do } x := x+1, \sigma_{i+1} \rangle \Downarrow \sigma_0}$$

- We adapt the structural induction principle to work on the structure of derivations.

CMPS201 Lecture 4

49

### Induction on Derivations

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

1. For each derivation rule of the form

$$\frac{H_1 \dots H_n}{C}$$

2. Assume that  $P$  holds for a derivation of  $H_i$  ( $i = 1, \dots, n$ ).
3. Prove the the property holds for the derivation obtained from the derivations of  $H_i$  using the given rule.

CMPS201 Lecture 4

50

### Example of Induction on Derivations (I)

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

$$D :: \frac{}{\langle \text{skip}, \sigma \rangle \Downarrow \sigma}$$

- This means that  $c = \text{skip}$ , and  $\sigma' = \sigma$ .
- By inversion  $\langle c, \sigma \rangle \Downarrow \sigma'$  uses the rule for skip. Thus  $\sigma'' = \sigma$ .
- This is a base case in the induction.

CMPS201 Lecture 4

51

### Example of Induction on Derivations (II)

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

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

- 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.

CMPS201 Lecture 4

52

### Example of Induction on Derivations (III)

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

$$D :: \frac{D_1 :: \langle b, \sigma \rangle \Downarrow \text{true} \quad D_2 :: \langle c, \sigma \rangle \Downarrow \sigma_1 \quad D_3 :: \langle \text{while } b \text{ do } c, \sigma_1 \rangle \Downarrow \sigma'}{\langle \text{while } b \text{ do } c, \sigma \rangle \Downarrow \sigma'}$$

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

CMPS201 Lecture 4

53

### 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
  - we pick arbitrary  $x \in A$  and  $D :: P(x)$
  - and 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 lot of practice can help a lot!

CMPS201 Lecture 4

54

## Equivalence

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

$$e_1 \approx e_2 \text{ iff } \forall \sigma \in \Sigma. \forall n \in \mathbb{Z}. \langle e_1, \sigma \rangle \Downarrow n \text{ iff } \langle e_2, \sigma \rangle \Downarrow n$$

and for commands

$$c_1 \approx c_2 \text{ iff } \forall \sigma, \sigma' \in \Sigma. \langle c_1, \sigma \rangle \Downarrow \sigma' \text{ iff } \langle c_2, \sigma \rangle \Downarrow \sigma'$$

## Notes on Equivalence

- Equivalence is like validity:
  - It 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 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;  $c \approx c$
- $(x := e_1; x := e_2) \approx x := e_2$ . (When is this true?)
- while b do c  $\approx$  if b then c; while b do c else skip
- If  $e_1 \approx e_2$  then  $x := e_1 \approx x := e_2$
- while true do skip  $\approx$  while true do  $x := x + 1$
- If c is
  - while  $x \neq y$  do
  - if  $x \geq y$  then  $x := x - y$  else  $y := y - x$
 then  $(x := 221; y := 527; c) \approx (x := 17; y := 17)$

## Proving an Equivalence

- Prove that "skip;  $c \approx c$ " for all c.
- Assume that  $D :: \langle \text{skip}; c, \sigma \rangle \Downarrow \sigma'$ .
- By inversion (twice) we have that:

$$D :: \frac{\langle \text{skip}, \sigma \rangle \Downarrow \sigma' \quad D_1 :: \langle c, \sigma \rangle \Downarrow \sigma'}{\langle \text{skip}; c, \sigma \rangle \Downarrow \sigma'}$$

- Thus, we have  $D_1 :: \langle c, \sigma \rangle \Downarrow \sigma'$ .
- The other direction is similar.

## Proving an Inequivalence

- Prove that  $x := y \approx x := z$  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  $\langle x := y, \sigma \rangle \Downarrow \sigma[x := 0]$
- 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 (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