Chapter 1: Operational Semantics

Bohua Zhan

Institute of Software, Chinese Academy of Sciences

December 2023

Reading

- Concrete Semantics (Nipkow & Klein), Chapter 7
- Introduction to Formal Semantics (Zhou & Zhan), Chapter 1

Why study semantics

- To prove correctness of a program, we first need to rigorously define what a program *does*.
- In particular, the rules of Hoare logic that we will study later can be justified based on clearly defined operational or denotational semantics.

A simple programming language: IMP

$$
\begin{aligned}
com = {} & \mathbf{skip} && \text{(skip)} \\
\mid {} & var := aexp && \text{(assign)} \\
\mid {} & com;\ com && \text{(seq)} \\
\mid {} & \mathbf{if}\ bexp\ \mathbf{then}\ com\ \mathbf{else}\ com && \text{(if)} \\
\mid {} & \mathbf{while}\ bexp\ \mathbf{do}\ com && \text{(while)}
\end{aligned}
$$

- var: name of a variable
- $aexp$: arithmetic expression
- $bexp$: boolean expression

Example of a program

In the following, define

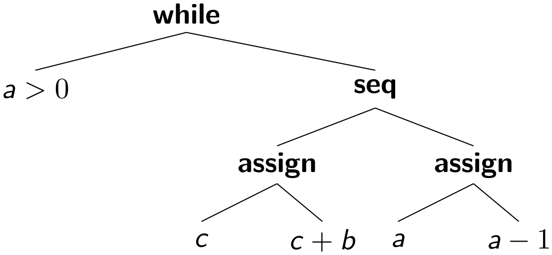
$$\begin{aligned} P \quad = \quad & \mathbf{while}\ a > 0\ \mathbf{do} \\ & \quad c := c + b\ ; \\ & \quad a := a - 1 \end{aligned}$$

Example of a program

In the following, define

$$
\begin{aligned}
P \quad = \quad & \mathbf{while}\ a > 0\ \mathbf{do} \\
& \quad c := c + b\ ; \\
& \quad a := a - 1
\end{aligned}
$$

Abstract syntax tree for P:



State

Semantics: what does the program P mean?

State

Semantics: what does the program P mean?

P represents a transformation on the state.

State

Semantics: what does the program P mean?

P represents a transformation on the state.

State: a function from variables to values (e.g. integers):

$$var \to int.$$

State

Semantics: what does the program P mean?

P represents a transformation on the state.

State: a function from variables to values (e.g. integers):

$$var \to int.$$

We assume the state is *finite*: it assigns values to only a finite number of variables.

Big-step operational semantics

- Given program c and states s, t,

$$(c, s) \Rightarrow t$$

means c can transform state s into state t.

Big-step operational semantics

- Given program c and states s, t,

$$(c, s) \Rightarrow t$$

  means c can transform state s into state t.

- Big-step operational semantics is specified by a set of rules for deriving statements of the above form.

Big-step operational semantics

- Given program c and states s, t,

$$(c, s) \Rightarrow t$$

  means c can transform state s into state t.

- Big-step operational semantics is specified by a set of rules for deriving statements of the above form.

- In the following, let $[\![e]\!]_s$ represent the evaluation of an arithmetic or boolean expression e on state s.

Rules for skip, assign and seq

- Rule for skip:

$$\frac{}{(\mathbf{skip}, s) \Rightarrow s}\ \mathsf{skip}$$

- Rule for assign:

$$\frac{}{(\mathsf{v} := \mathsf{e}, s) \Rightarrow s(v := [\![e]\!]_s)}\ \mathsf{assign}$$

- Rule for seq:

$$\frac{(c_1, s) \Rightarrow s' \quad (c_2, s') \Rightarrow s''}{(c_1; c_2, s) \Rightarrow s''}\ \mathsf{seq}$$

Rules for if

$$\frac{[\![b]\!]_s = \text{true} \quad (c_1, s) \Rightarrow t}{(\textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2,\ s) \Rightarrow t}\ \text{ifTrue}$$

$$\frac{[\![b]\!]_s = \text{false} \quad (c_2, s) \Rightarrow t}{(\textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2,\ s) \Rightarrow t}\ \text{ifFalse}$$

Rules for while

$$\frac{[\![b]\!]_s = \mathsf{false}}{(\mathbf{while}\ b\ \mathbf{do}\ c,\ s) \Rightarrow s}\ \mathsf{whileFalse}$$

$$\frac{[\![b]\!]_s = \mathsf{true} \quad (c, s) \Rightarrow s' \quad (\mathbf{while}\ b\ \mathbf{do}\ c,\ s') \Rightarrow s''}{(\mathbf{while}\ b\ \mathbf{do}\ c,\ s) \Rightarrow s''}\ \mathsf{whileTrue}$$

Operational semantics: example

Example: With

$$\begin{aligned} P \quad = \quad & \textbf{while } a > 0 \textbf{ do} \\ & \quad c := c + b \ ; \\ & \quad a := a - 1 \end{aligned}$$

show $(P, s) \Rightarrow t$ where

$$s = (\!| a := 2, b := 3, c := 0 |\!)$$
$$\text{and } t = (\!| a := 0, b := 3, c := 6 |\!).$$

First time through the loop

- Two assignments:

$$\dfrac{}{(c := c + b, (\!|a := 2, b := 3, c := 0|\!)) \Rightarrow (\!|a := 2, b := 3, c := 3|\!)}\ \text{assign}$$

$$\dfrac{}{(a := a - 1, (\!|a := 2, b := 3, c := 3|\!)) \Rightarrow (\!|a := 1, b := 3, c := 3|\!)}\ \text{assign}$$

- Combining:

$$\dfrac{\begin{array}{c}(c := c + b, (\!|a := 2, b := 3, c := 0|\!)) \Rightarrow (\!|a := 2, b := 3, c := 3|\!) \\ (a := a - 1, (\!|a := 2, b := 3, c := 3|\!)) \Rightarrow (\!|a := 1, b := 3, c := 3|\!)\end{array}}{(c := c + b; a := a - 1, (\!|a := 2, b := 3, c := 0|\!)) \Rightarrow (\!|a := 1, b := 3, c := 3|\!)}\ \text{seq}$$

Second time through the loop

- Two assignments:

$$\frac{}{(c := c + b, (\!|a := 1, b := 3, c := 3|\!)) \Rightarrow (\!|a := 1, b := 3, c := 6|\!)}\ \text{assign}$$

$$\frac{}{(a := a - 1, (\!|a := 1, b := 3, c := 6|\!)) \Rightarrow (\!|a := 0, b := 3, c := 6|\!)}\ \text{assign}$$

- Combining:

$$\frac{\begin{array}{c}(c := c + b, (\!|a := 1, b := 3, c := 3|\!)) \Rightarrow (\!|a := 1, b := 3, c := 6|\!) \\ (a := a - 1, (\!|a := 1, b := 3, c := 6|\!)) \Rightarrow (\!|a := 0, b := 3, c := 6|\!)\end{array}}{(c := c + b; a := a - 1, (\!|a := 1, b := 3, c := 3|\!)) \Rightarrow (\!|a := 0, b := 3, c := 6|\!)}\ \text{seq}$$

Dealing with **while**

Let D stand for $c := c + b; a := a - 1$

$$\frac{[\![a > 0]\!]_s = \mathsf{false}}{(\textbf{while } a > 0 \textbf{ do } D, (\!|a := 0, b := 3, c := 6|\!)) \Rightarrow (\!|a := 0, b := 3, c := 6|\!)}\ \mathsf{whileFalse}$$

$$\frac{\begin{array}{c}[\![a > 0]\!]_s = \mathsf{true} \\ (D, (\!|a := 1, b := 3, c := 3|\!)) \Rightarrow (\!|a := 0, b := 3, c := 6|\!) \\ (\textbf{while } a > 0 \textbf{ do } D, (\!|a := 0, b := 3, c := 6|\!)) \Rightarrow (\!|a := 0, b := 3, c := 6|\!)\end{array}}{(\textbf{while } a > 0 \textbf{ do } D, (\!|a := 1, b := 3, c := 3|\!)) \Rightarrow (\!|a := 0, b := 3, c := 6|\!)}\ \mathsf{whileTrue}$$

$$\frac{\begin{array}{c}[\![a > 0]\!]_s = \mathsf{true} \\ (D, (\!|a := 2, b := 3, c := 0|\!)) \Rightarrow (\!|a := 1, b := 3, c := 3|\!) \\ (\textbf{while } a > 0 \textbf{ do } D, (\!|a := 1, b := 3, c := 3|\!)) \Rightarrow (\!|a := 0, b := 3, c := 6|\!)\end{array}}{(\textbf{while } a > 0 \textbf{ do } D, (\!|a := 2, b := 3, c := 0|\!)) \Rightarrow (\!|a := 0, b := 3, c := 6|\!)}\ \mathsf{whileTrue}$$

Theoretical properties

- Termination
- Determinism
- Equivalence of commands

Termination

Using big-step operational semantics for IMP:

定义

A program c terminates on state s if there exists t such that $(c, s) \Rightarrow t$.

Note: in the IMP language there is no *failure*. If there is, we would need to somehow distinguish it from non-termination.

Determinism

Intuitively: a language is deterministic if any program starting at any state has at most one execution path.

Formal definition using big-step operational semantics:

定义

A language is deterministic if for any program c and state s, we have $(c, s) \Rightarrow t_1$ and $(c, s) \Rightarrow t_2$ implies $t_1 = t_2$.

Note: here we are ignoring possibility of non-termination.

Equivalence of commands

Intuitively: two commands are equivalent if their behaviors are the same.

Formal definition using big-step operational semantics:

定义

$c \sim c'$ if $\forall s\ t.\ (c, s) \Rightarrow t \longleftrightarrow (c', s) \Rightarrow t$.

Note: again, we ignore the possibility of non-termination.

Problems with big-step operational semantics

Cannot get any information about intermediate states

Problems with big-step operational semantics

Cannot get any information about intermediate states

. . . as a result . . .

Problems with big-step operational semantics

Cannot get any information about intermediate states

... as a result ...

- If P is non-terminating, cannot get any information about P at all. In particular, cannot distinguish non-termination and failure.

Problems with big-step operational semantics

Cannot get any information about intermediate states

... as a result ...

- If P is non-terminating, cannot get any information about P at all. In particular, cannot distinguish non-termination and failure.
- Unable to extend to concurrent execution, which involves interleaved execution of multiple threads.

- Big-step operational semantics (also called *natural semantics*) has the advantage of being simple, but as we have seen also has problems.

Small-step operational semantics

- Big-step operational semantics (also called *natural semantics*) has the advantage of being simple, but as we have seen also has problems.
- Next, we introduce small-step operational semantics (also called *structural operational semantics*), which addresses these problems.

Small-step operational semantics

- Big-step operational semantics (also called *natural semantics*) has the advantage of being simple, but as we have seen also has problems.
- Next, we introduce small-step operational semantics (also called *structural operational semantics*), which addresses these problems.
- Given programs c, c' and states s, t,

$$(c, s) \to (c', t)$$

  means execution of c on state s for one step can result in state t, with c' being the remainder of the program to be executed.

Rules for skip and assign

- There is *no* rule for **skip**, indicating that if the program is **skip**, then there is no more step to be taken.
- Rule for assign:

$$\frac{}{(v := e, s) \to (\mathbf{skip}, s(v := [\![e]\!]_s))}\ \text{assign}$$

If the program is an assignment, the next step performs the assignment, and there is nothing remaining to execute.

Rules for seq

$$\frac{(c_1, s) \to (c_1', s')}{(c_1; c_2, s) \to (c_1'; c_2, s')}\ \text{seq1}$$

$$\frac{}{(\textbf{skip}; c_2, s) \to (c_2, s)}\ \text{seq2}$$

To execute a sequence $c_1; c_2$, first execute c_1 (rule seq1). After c_1 has finished (becomes **skip**), start executing c_2 (rule seq2).

Rules for if

$$\frac{[\![b]\!]_s = \text{true}}{(\textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2,\ s) \to (c_1, s)}\ \text{ifTrue}$$

$$\frac{[\![b]\!]_s = \text{false}}{(\textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2,\ s) \to (c_2, s)}\ \text{ifFalse}$$

Rule for while

$$\frac{}{(\textbf{while}\ b\ \textbf{do}\ c, s) \rightarrow (\textbf{if}\ b\ \textbf{then}\ (c; \textbf{while}\ b\ \textbf{do}\ c)\ \textbf{else}\ \textbf{skip}, s)}\ \text{while}$$

Executing **while** for one step means unfolding the loop.

Execution in small-step semantics

定义

Define $\rightarrow^*$ to be the *reflexive transitive closure* of $\rightarrow$.

Intuitively:

- $(c, s) \rightarrow^* (c', s')$ means executing c on state s for some number of (including zero) steps results in a state s', with c' remaining to be executed.
- $(c, s) \rightarrow^* (\mathbf{skip}, s')$ means executing c on state s terminates in a state s'.

Example of execution

Let D stand for $c := c + b; a := a - 1$

$$
\begin{aligned}
& (\textbf{while } a > 0 \textbf{ do } D, (\!| a := 2, b := 3, c := 0 |\!)) \\
\to\ & (\textbf{if } a > 0 \textbf{ then } (D; \textbf{while } a > 0 \textbf{ do } D) \textbf{ else skip}, (\!| a := 2, b := 3, c := 0 |\!)) \\
\to\ & (D; \textbf{while } a > 0 \textbf{ do } D, (\!| a := 2, b := 3, c := 0 |\!)) \\
\to\ & (\textbf{skip}; a := a - 1; \textbf{while } a > 0 \textbf{ do } D, (\!| a := 2, b := 3, c := 3 |\!)) \\
\to\ & (a := a - 1; \textbf{while } a > 0 \textbf{ do } D, (\!| a := 2, b := 3, c := 3 |\!)) \\
\to\ & (\textbf{skip}; \textbf{while } a > 0 \textbf{ do } D, (\!| a := 1, b := 3, c := 3 |\!)) \\
\to\ & (\textbf{while } a > 0 \textbf{ do } D, (\!| a := 1, b := 3, c := 3 |\!))
\end{aligned}
$$

Example of execution

Let D stand for $c := c + b; a := a - 1$

$$
\begin{aligned}
& (\textbf{while } a > 0 \textbf{ do } D, (\!| a := 2, b := 3, c := 0 |\!)) \\
\to\ & (\textbf{if } a > 0 \textbf{ then } (D; \textbf{while } a > 0 \textbf{ do } D) \textbf{ else skip}, (\!| a := 2, b := 3, c := 0 |\!)) \\
\to\ & (D; \textbf{while } a > 0 \textbf{ do } D, (\!| a := 2, b := 3, c := 0 |\!)) \\
\to\ & (\textbf{skip}; a := a - 1; \textbf{while } a > 0 \textbf{ do } D, (\!| a := 2, b := 3, c := \color{red}{3} |\!)) \\
\to\ & (a := a - 1; \textbf{while } a > 0 \textbf{ do } D, (\!| a := 2, b := 3, c := 3 |\!)) \\
\to\ & (\textbf{skip}; \textbf{while } a > 0 \textbf{ do } D, (\!| a := \color{red}{1}, b := 3, c := 3 |\!)) \\
\to\ & (\textbf{while } a > 0 \textbf{ do } D, (\!| a := 1, b := 3, c := 3 |\!))
\end{aligned}
$$

Exercise: justify each of the above steps.

Example of execution, continued

$$
\begin{aligned}
&\dots \\
\rightarrow\ &(\textbf{while}\ a > 0\ \textbf{do}\ D, (\!|a := 1, b := 3, c := 3|\!)) \\
\rightarrow\ &(\textbf{if}\ a > 0\ \textbf{then}\ (D; \textbf{while}\ a > 0\ \textbf{do}\ D)\ \textbf{else skip}, (\!|a := 1, b := 3, c := 3|\!)) \\
\rightarrow\ &(D; \textbf{while}\ a > 0\ \textbf{do}\ D, (\!|a := 1, b := 3, c := 3|\!)) \\
\rightarrow\ &(\textbf{skip}; a := a - 1; \textbf{while}\ a > 0\ \textbf{do}\ D, (\!|a := 1, b := 3, c := {\color{red}6}|\!)) \\
\rightarrow\ &(a := a - 1; \textbf{while}\ a > 0\ \textbf{do}\ D, (\!|a := 1, b := 3, c := 6|\!)) \\
\rightarrow\ &(\textbf{skip}; \textbf{while}\ a > 0\ \textbf{do}\ D, (\!|a := {\color{red}0}, b := 3, c := 6|\!)) \\
\rightarrow\ &(\textbf{while}\ a > 0\ \textbf{do}\ D, (\!|a := 0, b := 3, c := 6|\!)) \\
\rightarrow\ &(\textbf{if}\ a > 0\ \textbf{then}\ (D; \textbf{while}\ a > 0\ \textbf{do}\ D)\ \textbf{else skip}, (\!|a := 0, b := 3, c := 6|\!)) \\
\rightarrow\ &(\textbf{skip}, (\!|a := 0, b := 3, c := 6|\!))
\end{aligned}
$$

Example of execution, continued

$$
\begin{aligned}
&\ldots \\
\to\ &(\textbf{while } a > 0 \textbf{ do } D, (\!|a := 1, b := 3, c := 3|\!)) \\
\to\ &(\textbf{if } a > 0 \textbf{ then } (D; \textbf{while } a > 0 \textbf{ do } D) \textbf{ else skip}, (\!|a := 1, b := 3, c := 3|\!)) \\
\to\ &(D; \textbf{while } a > 0 \textbf{ do } D, (\!|a := 1, b := 3, c := 3|\!)) \\
\to\ &(\textbf{skip}; a := a - 1; \textbf{while } a > 0 \textbf{ do } D, (\!|a := 1, b := 3, c := 6|\!)) \\
\to\ &(a := a - 1; \textbf{while } a > 0 \textbf{ do } D, (\!|a := 1, b := 3, c := 6|\!)) \\
\to\ &(\textbf{skip}; \textbf{while } a > 0 \textbf{ do } D, (\!|a := 0, b := 3, c := 6|\!)) \\
\to\ &(\textbf{while } a > 0 \textbf{ do } D, (\!|a := 0, b := 3, c := 6|\!)) \\
\to\ &(\textbf{if } a > 0 \textbf{ then } (D; \textbf{while } a > 0 \textbf{ do } D) \textbf{ else skip}, (\!|a := 0, b := 3, c := 6|\!)) \\
\to\ &(\textbf{skip}, (\!|a := 0, b := 3, c := 6|\!))
\end{aligned}
$$

Hence:

$$(\textbf{while } a > 0 \textbf{ do } D, (\!|a := 2, b := 3, c := 0|\!)) \to^* (\textbf{skip}, (\!|a := 0, b := 3, c := 6|\!))$$

Termination

- A program c *possibly* terminates on state s if there exists t such that
$$(c, s) \to^* (\mathbf{skip}, t)$$
(that is, there exists a finite path of $\to$ from (c, s) to $(\mathbf{skip}, t)$).
- A program c is *possibly* non-terminating on state s if there exists an infinite path of $\to$ starting from (c, s).
- A program c *always* terminates on state s if there does not exist an infinite path of $\to$ starting from (c, s).
- A program c stops abnormally if neither of the above is true. That is, there is a finite path of $\to$ from (c, s) to (c', t), where $c' \neq \mathbf{skip}$ but it is impossible to continue from (c', t).

Determinism

定义

A language is deterministic if for any program c and state s, we have $(c, s) \to (c_1, t_1)$ and $(c, s) \to (c_2, t_2)$ implies $c_1 = c_2$ and $t_1 = t_2$.

Note: this corresponds to our intuitive concept of deterministic execution. It also takes into account the possibility of non-termination.

Equivalence of big- and small-step semantics

定理 (from big to small)

$(c, s) \Rightarrow t$ *implies* $(c, s) \rightarrow^* (\mathbf{skip}, t)$.

定理 (from small to big)

$(c, s) \rightarrow^* (\mathbf{skip}, t)$ *implies* $(c, s) \Rightarrow t$.

Proofs by induction

How to prove theorems in the theory of operational semantics?

Proofs by induction

How to prove theorems in the theory of operational semantics?

- Induction on structure of the program: when proving a property about all programs, assume the property holds on sub-programs (in the sense of abstract syntax tree).

How to prove theorems in the theory of operational semantics?

- Induction on structure of the program: when proving a property about all programs, assume the property holds on sub-programs (in the sense of abstract syntax tree).
- Induction on proof tree: when proving a consequence from an assertion of an inductively defined predicate, induct on how the assertion is obtained.

Example of proof

We aim to prove the following theorem:

定理

$(c, s) \Rightarrow t$ *implies* $(c, s) \rightarrow^* (\mathbf{skip}, t)$.

Example of proof

We aim to prove the following theorem:

定理

$(c, s) \Rightarrow t$ *implies* $(c, s) \rightarrow^* (\mathbf{skip}, t)$.

For this, we need a lemma about small-step semantics:

引理

$(c_1, s_1) \rightarrow^* (c_2, s_2)$ *implies* $(c_1; c', s_1) \rightarrow^* (c_2; c', s_2)$.

Example of proof

We aim to prove the following theorem:

定理

$(c, s) \Rightarrow t$ *implies* $(c, s) \rightarrow^* (\mathbf{skip}, t)$.

For this, we need a lemma about small-step semantics:

引理

$(c_1, s_1) \rightarrow^* (c_2, s_2)$ *implies* $(c_1; c', s_1) \rightarrow^* (c_2; c', s_2)$.

Recall the definition of reflexive transitive closure...

Proof of lemma

$$\frac{}{(c, s) \to^* (c, s)} \qquad \frac{(c_1, s_1) \to (c_3, s_3) \quad (c_3, s_3) \to^* (c_2, s_2)}{(c_1, s_1) \to^* (c_2, s_2)}$$

Proof (of lemma):

- Base case: we have $(c, s) \to^* (c, s)$ and need to show $(c; c', s) \to^* (c; c', s)$. This follows from reflexivity of the closure.
- Inductive step: we have $(c_1, s_1) \to (c_3, s_3)$ and $(c_3, s_3) \to^* (c_2, s_2)$, and wish to show $(c_1; c', s_1) \to^* (c_2; c', s_2)$.
  - By rule seq1 of small step semantics, we have $(c_1; c', s_1) \to^* (c_3; c', s_3)$.
  - By inductive hypothesis, we have $(c_3; c', s_3) \to^* (c_2; c', s_2)$.
  - By the transitivity of the closure, we have $(c_1; c', s_1) \to^* (c_2; c', s_2)$, as desired.

Proof of theorem

Proof (of theorem): induct on proof of $(c, s) \Rightarrow t$, divide into cases by the last rule used.

- skip: then $c = \mathbf{skip}$ and $s = t$, we wish to show

$$(\mathbf{skip}, s) \to^* (\mathbf{skip}, s).$$

  This follows from reflexivity.

Proof of theorem

Proof (of theorem): induct on proof of $(c, s) \Rightarrow t$, divide into cases by the last rule used.

- skip: then $c = \mathbf{skip}$ and $s = t$, we wish to show

  $$(\mathbf{skip}, s) \to^* (\mathbf{skip}, s).$$

  This follows from reflexivity.

- assign: then $c = (v := e)$ and $t = s(v := [\![e]\!]_s)$, we wish to show

  $$(v := e, s) \to^* (\mathbf{skip}, s(v := [\![e]\!]_s)).$$

  This follows from rule assign for small-step semantics.

Proof of theorem (continued)

- seq: then $c = c_1; c_2$, and there is some state s' such that $(c_1, s) \Rightarrow s'$ and $(c_2, s') \Rightarrow t$. By inductive hypothesis, we have $(c_1, s) \rightarrow^* (\textbf{skip}, s')$ and $(c_2, s') \rightarrow^* (\textbf{skip}, t)$. By the previous lemma, we have $(c_1; c_2, s) \rightarrow^* (\textbf{skip}; c_2, s')$. Combining these with rule seq2, we get $(c_1; c_2, s) \rightarrow^* (\textbf{skip}, t)$.

Proof of theorem (continued)

- seq: then $c = c_1; c_2$, and there is some state s' such that $(c_1, s) \Rightarrow s'$ and $(c_2, s') \Rightarrow t$. By inductive hypothesis, we have $(c_1, s) \to^* (\mathbf{skip}, s')$ and $(c_2, s') \to^* (\mathbf{skip}, t)$. By the previous lemma, we have $(c_1; c_2, s) \to^* (\mathbf{skip}; c_2, s')$. Combining these with rule seq2, we get $(c_1; c_2, s) \to^* (\mathbf{skip}, t)$.
- ifTrue: then c is of the form $\mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2$, $[\![b]\!]_s = \text{true}$, and $(c_1, s) \Rightarrow t$. By inductive hypothesis, we get $(c_1, s) \to^* (\mathbf{skip}, t)$, Combine this with rule ifTrue for small-step semantics, we get $(c, s) \to^* (\mathbf{skip}, t)$ as desired.
- ifFalse: similar to ifTrue case.

Proof of theorem (continued)

- whileFalse: then c is of the form $\mathbf{while}\ b\ \mathbf{do}\ c'$, $[\![b]\!]_s = \text{false}$, and $s = t$. By rule while followed by ifFalse, we get $(c, s) \to^* (\mathbf{skip}, t)$, as desired.

Proof of theorem (continued)

- whileFalse: then c is of the form $\mathbf{while}\ b\ \mathbf{do}\ c'$, $[\![b]\!]_s = \text{false}$, and $s = t$. By rule while followed by ifFalse, we get $(c, s) \to^* (\mathbf{skip}, t)$, as desired.
- whileTrue: then c is of the form $\mathbf{while}\ b\ \mathbf{do}\ c'$, $[\![b]\!]_s = \text{true}$, $(c', s) \Rightarrow s'$ and $(c, s') \Rightarrow t$. By the inductive hypothesis, we get $(c', s) \to^* (\mathbf{skip}, s')$ and $(c, s') \to^* (\mathbf{skip}, t)$. By the previous lemma, we get $(c'; c, s) \to^* (\mathbf{skip}; c, s')$. Combining these with while, ifTrue and seq2, we get $(c, s) \to^* (\mathbf{skip}, t)$, as desired.

What have we learned?

- We proved (one direction of) the equivalence between two definitions of semantics.

What have we learned?

- We proved (one direction of) the equivalence between two definitions of semantics.
- This is possible only because both semantics are defined in a precise way (using a logical language).

What have we learned?

- We proved (one direction of) the equivalence between two definitions of semantics.
- This is possible only because both semantics are defined in a precise way (using a logical language).
- The proofs are tedious and error-prone. An interactive theorem prover will help a lot in checking such proofs.

Extensions

- **for** loops, **do** . . . **while** . . . loops.
- Jumps (**goto**s).
- Local variables (scopes).
- Procedures (pushing and popping stack).
- Exceptions (throw and catch).
- Data types, structs, pointers (aliasing), arrays.
- Objects, classes, methods (inheritance and polymorphism).

Extensions

- **for** loops, **do** ... **while** ... loops.
- Jumps (**goto**s).
- Local variables (scopes).
- Procedures (pushing and popping stack).
- Exceptions (throw and catch).
- Data types, structs, pointers (aliasing), arrays.
- Objects, classes, methods (inheritance and polymorphism).

non-determinism, concurrency, randomization,

functional programming (closures), ...