Programming Language Theory

Big-step Operational Semantics (aka Natural Semantics)

Ralf Lämmel
A big-step operational semantics for While
Syntactic categories of the While language

- **numerals**
  \[ n \in \text{Num} \]

- **variables**
  \[ x \in \text{Var} \]

- **arithmetic expressions**
  \[ a \in \text{Aexp} \]
  \[ a ::= n \mid x \mid a_1 + a_2 \mid a_1 \times a_2 \mid a_1 - a_2 \]

- **booleans expressions**
  \[ b \in \text{Bexp} \]
  \[ b ::= \text{true} \mid \text{false} \mid a_1 = a_2 \mid a_1 \leq a_2 \mid \neg b \mid b_1 \land b_2 \]

- **statements**
  \[ S \in \text{Stm} \]
  \[ S ::= x := a \mid \text{skip} \mid S_1 ; S_2 \mid \text{if } b \text{ then } S_1 \text{ else } S_2 \mid \text{while } b \text{ do } S \]
Semantic categories of the \textbf{While} language

Natural numbers
\[ N = \{0, 1, 2, \ldots\} \]

Truth values
\[ T = \{tt, ff\} \]

States
\[ \text{State} = \text{Var} \to \mathbb{N} \]

Lookup in a state: \[ s \ x \]

Update a state: \[ s' = s[y \mapsto v] \]

\[ s' \ x = \begin{cases} s \ x & \text{if } x \neq y \\ v & \text{if } x = y \end{cases} \]
Meanings of syntactic categories

Numerals
\[ \mathcal{N} : \text{Num} \rightarrow \mathbb{N} \]

Variables
\[ s \in \text{State} = \text{Var} \rightarrow \mathbb{N} \]

Arithmetic expressions
\[ \mathcal{A} : \text{Aexp} \rightarrow (\text{State} \rightarrow \mathbb{N}) \]

Boolean expressions
\[ \mathcal{B} : \text{Bexp} \rightarrow (\text{State} \rightarrow \text{T}) \]

Statements
\[ \mathcal{S} : \text{Stm} \rightarrow (\text{State} \leftrightarrow \text{State}) \]

Semantics of arithmetic expressions

\[ \mathcal{A}[n]s = \mathcal{N}[n] \]
\[ \mathcal{A}[x]s = s \ x \]
\[ \mathcal{A}[a_1 + a_2]s = \mathcal{A}[a_1]s + \mathcal{A}[a_2]s \]
\[ \mathcal{A}[a_1 \cdot a_2]s = \mathcal{A}[a_1]s \cdot \mathcal{A}[a_2]s \]
\[ \mathcal{A}[a_1 - a_2]s = \mathcal{A}[a_1]s - \mathcal{A}[a_2]s \]
Semantics of boolean expressions

\[ B[\text{true}]s = \text{tt} \]
\[ B[\text{false}]s = \text{ff} \]
\[ B[a_1 = a_2]s = \begin{cases} 
\text{tt} & \text{if } A[a_1]s = A[a_2]s \\
\text{ff} & \text{if } A[a_1]s \neq A[a_2]s 
\end{cases} \]
\[ B[a_1 \leq a_2]s = \begin{cases} 
\text{tt} & \text{if } A[a_1]s \leq A[a_2]s \\
\text{ff} & \text{if } A[a_1]s \not\leq A[a_2]s 
\end{cases} \]
\[ B[\neg b]s = \begin{cases} 
\text{tt} & \text{if } B[b]s = \text{ff} \\
\text{ff} & \text{if } B[b]s = \text{tt} 
\end{cases} \]
\[ B[b_1 \land b_2]s = \begin{cases} 
\text{tt} & \text{if } B[b_1]s = \text{tt} \\
& \text{and } B[b_2]s = \text{tt} \\
\text{ff} & \text{if } B[b_1]s = \text{ff} \\
& \text{or } B[b_2]s = \text{ff} 
\end{cases} \]
Semantics of statements

\[\text{[ass}_n\text{s}] \quad \langle x := a, s \rangle \rightarrow s[x \rightarrow A[a]]s\]

\[\text{[skip}_n\text{s}] \quad \langle \text{skip}, s \rangle \rightarrow s\]

\[\text{[comp}_n\text{s}] \quad \frac{\langle S_1, s \rangle \rightarrow s', \langle S_2, s' \rangle \rightarrow s''}{\langle S_1; S_2, s \rangle \rightarrow s''}\]

\[\text{[if}_n\text{t}_t\text{]} \quad \frac{\langle S_1, s \rangle \rightarrow s'}{\langle \text{if } b \text{ then } S_1 \text{ else } S_2, s \rangle \rightarrow s'}\quad \text{if } B[b]s = \text{tt}\]

\[\text{[if}_n\text{f}_t\text{]} \quad \frac{\langle S_2, s \rangle \rightarrow s'}{\langle \text{if } b \text{ then } S_1 \text{ else } S_2, s \rangle \rightarrow s'}\quad \text{if } B[b]s = \text{ff}\]

\[\text{[while}_n\text{t}_t\text{]} \quad \frac{\langle S, s \rangle \rightarrow s', \langle \text{while } b \text{ do } S, s' \rangle \rightarrow s''}{\langle \text{while } b \text{ do } S, s \rangle \rightarrow s''}\quad \text{if } B[b]s = \text{tt}\]

\[\text{[while}_n\text{f}_f\text{]} \quad \langle \text{while } b \text{ do } S, s \rangle \rightarrow s \quad \text{if } B[b]s = \text{ff}\]
Derivation trees

\[
\begin{align*}
\langle z:=x, \ s_0 \rangle & \rightarrow s_1 & \langle x:=y, \ s_1 \rangle & \rightarrow s_2 \\
\hline
\langle z:=x; \ x:=y, \ s_0 \rangle & \rightarrow s_2 & \langle y:=z, \ s_2 \rangle & \rightarrow s_3 \\
\hline
\langle z:=x; \ x:=y; \ y:=z, \ s_0 \rangle & \rightarrow s_3
\end{align*}
\]

States

\[
\begin{align*}
s_0 & = [x\rightarrow 5, \ y\rightarrow 7, \ z\rightarrow 0] \\
s_1 & = [x\rightarrow 5, \ y\rightarrow 7, \ z\rightarrow 5] \\
s_2 & = [x\rightarrow 7, \ y\rightarrow 7, \ z\rightarrow 5] \\
s_3 & = [x\rightarrow 7, \ y\rightarrow 5, \ z\rightarrow 5]
\end{align*}
\]
Prolog as a sandbox for big-step operational semantics

https://slps.svn.sourceforge.net/svnroot/slps/topics/NielsonN07/Prolog/While/NS/
Architecture of the interpreter

- **Makefile**: see “make test”
- **main.pro**: main module to compose all other modules
- **exec.pro**: statement execution
- **eval.pro**: expression evaluation
- **map.pro**: abstract data type for maps (states)
- **test.pro**: framework for unit testing
main.pro

:- ['eval.pro'].
:- ['exec.pro'].
:- ['map.pro'].
:- ['test.pro'].

% Tests

:- test(evala(add(num(21),num(21)),_,42)).
...
:- halt.
Tests

:- test(evala(add(num(21),num(21)),_,42)).
:- test(evala(add(num(21),id(x)),['x',21],42)).
:- test(
    exec(
        while( not(eq(id(x),num(0))),
            seq(
                assign(y,mul(id(x),id(y))),
                assign(x,sub(id(x),num(1)))),
        [(x,5),(y,1)],
        [(x,0),(y,120)]).
)
Arithmetic expression evaluation

% Number is evaluated to its value
evala(num(V),_,V).

% Variable reference is evaluated to its current value
evala(id(X),M,Y) :~ lookup(M,X,Y).
Arithmetic expression evaluation cont’d

% Addition
evala(add(A1,A2),M,V) :-
  evala(A1,M,V1),
  evala(A2,M,V2),
  V is V1 + V2.

% Subtraction
...

% Multiplication
...
Boolean expression evaluation

\[
\text{evalb}(true,\_, tt).
\]
\[
\text{evalb}(false,\_, ff).
\]
\[
\text{evalb}(\text{not}(B), M, V) :-
\text{evalb}(B, M, V1),
\text{not}(V1, V).
\]
\[
\text{evalb}(\text{and}(B1, B2), M, V) :-
\text{evalb}(B1, M, V1),
\text{evalb}(B2, M, V2),
\text{and}(V1, V2, V).
\]
\[
\vdots
\]

© Ralf Lämmel, 2009-2012 unless noted otherwise
Skip statement

exec(skip,M,M).
Sequential composition

\[
\text{exec(seq(S1,S2),M1,M3) :- exec(S1,M1,M2), exec(S2,M2,M3).}
\]
Assignment

exec(assign(X,A),M1,M2) :-
evala(A,M1,Y),
update(M1,X,Y,M2).
Conditional

% Conditional statement with true condition
exec(ifthenelse(B,S1,_),M1,M2) :-
  evalb(B,M1,tt),
  exec(S1,M1,M2).

% Conditional statement with false condition
exec(ifthenelse(B,_,S2),M1,M2) :-
  evalb(B,M1,ff),
  exec(S2,M1,M2).
Loop statement

% Loop statement with true condition
exec(while(B,S),M1,M3) :-
   evalb(B,M1,tt),
   exec(S,M1,M2),
   exec(while(B,S),M2,M3).

% Loop statement with false condition
exec(while(B,_),M,M) :-
   evalb(B,M,ff).
Abstract data type for maps (states)

% Function lookup (application)
lookup(M,X,Y) :- append(_,[(X,Y)|_],M).

% Function update in one position
update([],X,Y,[X,Y]).
update([X,Y|M],X,Y,[X,Y|M]).
update([(X1,Y1)|M1],X2,Y2,[X1,Y1]|M2) :- 
    X1 = X2,
    update(M1,X2,Y2,M2).
Test framework

test(G)
:-
    ( G -> P = 'OK'; P = 'FAIL' ),
    format('~w: ~w~n', [P,G]).
Blocks and procedures

\[ S ::= x := a \mid \text{skip} \mid S_1 ; S_2 \]
\[ \quad | \quad \text{if } b \text{ then } S_1 \text{ else } S_2 \]
\[ \quad | \quad \text{while } b \text{ do } S \]
\[ \quad | \quad \text{begin } D_V D_P S \text{ end} \]
\[ \quad | \quad \text{call } p \]

\[ D_V ::= \text{var } x := a; D_V \mid \epsilon \]
\[ D_P ::= \text{proc } p \text{ is } S; D_P \mid \epsilon \]
Semantics of var declarations

Extension of semantics of statements:

\[
(D_V, s) \rightarrow_D s', (S, s') \rightarrow s'' \\
\text{(begin } D_V S \text{ end, } s) \rightarrow s''[DV(D_V)\mapsto s]
\]

Semantics of variable declarations:

\[
(D_V, s[x\mapsto A[a]\{s\}]) \rightarrow_D s' \\
(\text{var } x := a; D_V, s) \rightarrow_D s' \\
(\varepsilon, s) \rightarrow_D s
\]
Scope rules

- Dynamic scope for variables and procedures
- Dynamic scope for variables but static for procedures
- Static scope for variables as well as procedures

```
begin var x := 0;
    proc p is x := x * 2;
    proc q is call p;
    begin var x := 5;
        proc p is x := x + 1;
        call q; y := x
    end
end
```
Dynamic scope for variables and procedures

• Execution
  ✦ call q
  ✦ call p (calls inner, say local p)
  ✦ \( x := x + 1 \) (affects inner, say local \( x \))
  ✦ \( y := x \) (obviously accesses local \( x \))

• Final value of \( y = 6 \)

\[
\begin{align*}
\text{[ass]} & \quad e w_P \vdash \langle x := a, s \rangle \rightarrow s[x \mapsto A[a]]s \\
\text{[skip]} & \quad e w_P \vdash \langle \text{skip}, s \rangle \rightarrow s \\
\text{[comp]} & \quad \frac{e w_P \vdash \langle S_1, s \rangle \rightarrow s', \ e w_P \vdash \langle S_2, s' \rangle \rightarrow s''}{e w_P \vdash \langle S_1; S_2, s \rangle \rightarrow s''} \\
\text{[if\_t]} & \quad \frac{e w_P \vdash \langle \text{if } b \text{ then } S_1 \text{ else } S_2, s \rangle \rightarrow s'}{e w_P \vdash \langle S_1, s \rangle \rightarrow s'} \\
\text{if}\_t & \quad \text{if } B[b]s = \text{tt} \\
\text{[if\_f]} & \quad \frac{e w_P \vdash \langle \text{if } b \text{ then } S_1 \text{ else } S_2, s \rangle \rightarrow s'}{e w_P \vdash \langle S_2, s \rangle \rightarrow s'} \\
\text{if}\_f & \quad \text{if } B[b]s = \text{ff} \\
\text{[while\_t]} & \quad \frac{e w_P \vdash \langle S, s \rangle \rightarrow s', \ e w_P \vdash \langle \text{while } b \text{ do } S, s' \rangle \rightarrow s''}{e w_P \vdash \langle \text{while } b \text{ do } S, s \rangle \rightarrow s''} \\
\text{while}\_t & \quad \text{if } B[b]s = \text{tt} \\
\text{[while\_f]} & \quad e w_P \vdash \langle \text{while } b \text{ do } S, s \rangle \rightarrow s \\
\text{while}\_f & \quad \text{if } B[b]s = \text{ff} \\
\text{[block]} & \quad \langle D_V, s \rangle \rightarrow D s', \ \text{upd}_P(D_P, e w_P) \vdash \langle S, s' \rangle \rightarrow s'' \\
\text{[call]} & \quad \text{upd}_P(\langle \text{call } p, s \rangle \rightarrow s') \quad \text{where } e w_P p = S \\
\text{upd}_P(\langle \text{proc } p \text{ is } S; D_P, e w_P \rangle) = \text{upd}_P(D_P, e w_P[p \mapsto S]) \\
\text{upd}_P(\epsilon, e w_P) = e w_P
\end{align*}
\]

NS with dynamic scope rules using an environment

\[\text{Env}_P = \text{Pname} \rightarrow \text{Stm}\]
Dynamic scope for variables
Static scope for procedures

• Execution
  ✦ call q
  ✦ call p (calls outer, say global p)
  ✦ x := x * 2 (affects inner, say local x)
  ✦ y := x (obviously accesses local x)

• Final value of y = 10

begin var x := 0;
  proc p is x := x * 2;
  proc q is call p;
  begin var x := 5;
    proc p is x := x + 1;
    call q; y := x
  end
end
Dynamic scope for variables
Static scope for procedures

- Updated environment
  \[ \text{Env}_P = \text{Pname} \leftrightarrow \text{Stm} \times \text{Env}_P \]

- Updated environment update
  \[
  \text{upd}_P(\text{proc } p \text{ is } S; D_P, \text{env}_P) = \text{upd}_P(D_P, \text{env}_P[p\rightarrow(S; \text{env}_P)])
  \]
  \[
  \text{upd}_P(\varepsilon, \text{env}_P) = \text{env}_P
  \]

- Updated rule for calls
  \[
  \text{env}_P' \vdash \langle S, s \rangle \rightarrow s'
  \]

  \[
  \text{env}_P \vdash \langle \text{call } p, s \rangle \rightarrow s'
  \]

  \[
  \text{where } \text{env}_P' p = (S, \text{env}_P')
  \]

- Recursive calls
  \[
  \text{env}_P'[p\rightarrow(S, \text{env}_P')] \vdash \langle S, s \rangle \rightarrow s'
  \]

  \[
  \text{env}_P \vdash \langle \text{call } p, s \rangle \rightarrow s'
  \]

  \[
  \text{where } \text{env}_P' p = (S, \text{env}_P')
  \]
Static scope for variables and procedures

• Execution

✧ call q

✧ call p (calls outer, say global p)

✧ x := x * 2 (affects outer, say global x)

✧ y := x (obviously accesses local x)

• Final value of y = 5

begin var x := 0;
  proc p is x := x * 2;
  proc q is call p;
  begin var x := 5;
    proc p is x := x + 1;
    call q; y := x
  end
end

Formal semantics omitted here.
Properties of semantics and induction proofs
One property of the semantics

**Lemma [1.11]**

Let $s$ and $s'$ be two states satisfying

$s \ x = s' \ x$

for all $x \in \text{FV}(a)$. Then

$A[a]s = A[a]s'$

Intuitively: The value of an arithmetic expression only depends on the values of the variables that occur in it.

<table>
<thead>
<tr>
<th>Free variables in arithmetic expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{FV}(n) = \emptyset$</td>
</tr>
<tr>
<td>$\text{FV}(x) = {x}$</td>
</tr>
<tr>
<td>$\text{FV}(a_1 + a_2) = \text{FV}(a_1) \cup \text{FV}(a_2)$</td>
</tr>
<tr>
<td>$\text{FV}(a_1 \ast a_2) = \text{FV}(a_1) \cup \text{FV}(a_2)$</td>
</tr>
<tr>
<td>$\text{FV}(a_1 - a_2) = \text{FV}(a_1) \cup \text{FV}(a_2)$</td>
</tr>
</tbody>
</table>

Proof by structural induction on the arithmetic expressions
Consider again the semantics of arithmetic expressions

\[
\begin{align*}
A[n]s &= N[n] \\
A[x]s &= s x \\
A[a_1 + a_2]s &= A[a_1]s + A[a_2]s \\
A[a_1 \ast a_2]s &= A[a_1]s \ast A[a_2]s \\
A[a_1 - a_2]s &= A[a_1]s - A[a_2]s
\end{align*}
\]

The definition obeys compositionality. Hence, induction on syntax is feasible.
Compositional Definitions

1: The syntactic category is specified by an abstract syntax giving the \textit{basis elements} and the \textit{composite elements}. The composite elements have a unique decomposition into their immediate constituents.

2: The semantics is defined by \textit{compositional} definitions of a function: There is a \textit{semantic clause} for each of the basis elements of the syntactic category and one for each of the methods for constructing composite elements. The clauses for composite elements are defined in terms of the semantics of the immediate constituents of the elements.
<table>
<thead>
<tr>
<th></th>
<th>Structural Induction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td>Prove that the property holds for all the <em>basis</em> elements of the syntactic category.</td>
</tr>
<tr>
<td>2:</td>
<td>Prove that the property holds for all the <em>composite</em> elements of the syntactic category: Assume that the property holds for all the immediate constituents of the element (this is called the <em>induction hypothesis</em>) and prove that it also holds for the element itself.</td>
</tr>
</tbody>
</table>
Let $s$ and $s'$ be two states satisfying
\[ s \ x = s' \ x \]
for all $x \in \text{FV}(a)$. Then
\[ A[a]s = A[a]s' \]

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A[n]s$</td>
<td>$N[n]$</td>
</tr>
<tr>
<td>$A[\times]s$</td>
<td>$s \ x$</td>
</tr>
<tr>
<td>$A[a_1 + a_2]s$</td>
<td>$A[a_1]s + A[a_2]s$</td>
</tr>
<tr>
<td>$A[a_1 \times a_2]s$</td>
<td>$A[a_1]s \times A[a_2]s$</td>
</tr>
<tr>
<td>$A[a_1 - a_2]s$</td>
<td>$A[a_1]s - A[a_2]s$</td>
</tr>
</tbody>
</table>

Table 1.1: The semantics of arithmetic expressions

## Proofs for basis elements

**The case $n$:** From Table 1.1 we have $A[n]s = N[n]$ as well as $A[n]s' = N[n]$. So $A[n]s = A[n]s'$ and clearly the lemma holds in this case.

**The case $x$:** From Table 1.1 we have $A[x]s = s \ x$ as well as $A[x]s' = s' \ x$. From the assumptions of the lemma we get $s \ x = s' \ x$ because $x \in \text{FV}(x)$ so clearly the lemma holds in this case.

Proof by structural induction on the arithmetic expressions
Let $s$ and $s'$ be two states satisfying
$$s \ x = s' \ x$$
for all $x \in \text{FV}(a)$. Then
$$A[a]s = A[a]s'$$

<table>
<thead>
<tr>
<th>$A[n]s$</th>
<th>$N[n]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A[x]s$</td>
<td>$s \ x$</td>
</tr>
<tr>
<td>$A[a_1 + a_2]s$</td>
<td>$A[a_1]s + A[a_2]s$</td>
</tr>
<tr>
<td>$A[a_1 \times a_2]s$</td>
<td>$A[a_1]s \times A[a_2]s$</td>
</tr>
<tr>
<td>$A[a_1 - a_2]s$</td>
<td>$A[a_1]s - A[a_2]s$</td>
</tr>
</tbody>
</table>

Table 1.1: The semantics of arithmetic expressions

**Proofs for composite elements**

**The case $a_1 + a_2$:** From Table 1.1 we have $A[a_1 + a_2]s = A[a_1]s + A[a_2]s$ and similarly $A[a_1 + a_2]s' = A[a_1]s' + A[a_2]s'$. Since $a_i$ (for $i = 1, 2$) is an immediate subexpression of $a_1 + a_2$ and $\text{FV}(a_i) \subseteq \text{FV}(a_1 + a_2)$ we can apply the induction hypothesis (that is the lemma) to $a_i$ and get $A[a_i]s = A[a_i]s'$. It is now easy to see that the lemma holds for $a_1 + a_2$ as well.

**The cases $a_1 - a_2$ and $a_1 \times a_2$** follow the same pattern and are omitted.
Another property of the semantics

**Theorem** [2.9] The natural semantics of While is deterministic, that is for all statements \( S \) of While and all states \( s, s', s'' \)

if \( (S, s) \rightarrow s' \) and \( (S, s) \rightarrow s'' \)
then \( s' = s'' \).

**Proof**

We assume \( (S, s) \rightarrow s' \).
We prove that if \( (S, s) \rightarrow s'' \) then \( s' = s'' \).

We proceed by induction on the inference of \( (S, s) \rightarrow s' \).
Induction on the shape of derivation trees

Basically, induction on the shape of derivation trees is a kind of structural induction on the derivation trees: In the base case we show that the property holds for the simple derivation trees. In the induction step we assume that the property holds for the immediate constituents of a derivation tree and show that it also holds for the composite derivation tree.

[while_{ns}^{tt}]

\[
\frac{(S, s) \rightarrow s', (\text{while } b \text{ do } S, s') \rightarrow s''}{(\text{while } b \text{ do } S, s) \rightarrow s''} \quad \text{if } B[b]s = \tt
\]

Structural induction on syntactical categories is \textbf{not} applicable because of the non-compositional semantics of while!
### Induction on the Shape of Derivation Trees

1. Prove that the property holds for all the simple derivation trees by showing that it holds for the *axioms* of the transition system.

2. Prove that the property holds for all composite derivation trees: For each *rule* assume that the property holds for its premises (this is called the *induction hypothesis*) and prove that it also holds for the conclusion of the rule provided that the conditions of the rule are satisfied.
The natural semantics of While is deterministic, that is for all statements $S$ of While and all states $s$, $s'$ and $s''$

if $(S, s) \rightarrow s'$ and $(S, s) \rightarrow s''$

then $s' = s''$.

**Proof:** We assume that $(S, s) \rightarrow s'$ and shall prove that

if $(S, s) \rightarrow s''$ then $s' = s''$.

We shall proceed by induction on the shape of the derivation tree for $(S, s) \rightarrow s'$.

**The case** $[[\text{ass}_{ns}]]$: Then $S$ is $x:=a$ and $s'$ is $s[x \leftarrow A[a]]s$. The only axiom or rule that could be used to give $(x:=a, s) \rightarrow s''$ is $[[\text{ass}_{ns}]]$ so it follows that $s''$ must be $s[x \leftarrow A[a]]s$ and thereby $s' = s''$.

**The case** $[[\text{skip}_{ns}]]$: Analogous.
**The case** \([\text{comp}_\text{ns}]\): Assume that

\[\langle S_1; S_2, s \rangle \rightarrow s'\]

holds because

\[\langle S_1, s \rangle \rightarrow s_0 \text{ and } \langle S_2, s_0 \rangle \rightarrow s'\]

for some \(s_0\). The only rule that could be applied to give \(\langle S_1; S_2, s \rangle \rightarrow s''\) is \([\text{comp}_\text{ns}]\) so there is a state \(s_1\) such that

\[\langle S_1, s \rangle \rightarrow s_1 \text{ and } \langle S_2, s_1 \rangle \rightarrow s''\]

The induction hypothesis can be applied to the premise \(\langle S_1, s \rangle \rightarrow s_0\) and from \(\langle S_1, s \rangle \rightarrow s_1\) we get \(s_0 = s_1\). Similarly, the induction hypothesis can be applied to the premise \(\langle S_2, s_0 \rangle \rightarrow s'\) and from \(\langle S_2, s_0 \rangle \rightarrow s''\) we get \(s' = s''\) as required.
The case \([\text{if}^{tt}]_{\text{ns}}\): Assume that

\[\langle \text{if } b \text{ then } S_1 \text{ else } S_2, s \rangle \rightarrow s'\]

holds because

\[\mathcal{B}[b] s = \text{tt} \text{ and } \langle S_1, s \rangle \rightarrow s'\]

From \(\mathcal{B}[b] s = \text{tt}\) we get that the only rule that could be applied to give the alternative \(\langle \text{if } b \text{ then } S_1 \text{ else } S_2, s \rangle \rightarrow s''\) is \([\text{if}^{tt}]_{\text{ns}}\). So it must be the case that

\[\langle S_1, s \rangle \rightarrow s''\]

But then the induction hypothesis can be applied to the premise \(\langle S_1, s \rangle \rightarrow s'\) and from \(\langle S_1, s \rangle \rightarrow s''\) we get \(s' = s''\).

The case \([\text{if}^{ff}]_{\text{ns}}\): Analogous.

Proof by induction on the shape of derivation trees
Non-compositional semantics is Ok for this proof scheme.

The case $\texttt{while}_\text{ns}^{tt}$: Analogous.

The case $\texttt{while}_\text{ns}^{ff}$: Straightforward.

Proof by induction on the shape of derivation trees
Yet another property of the semantics

**Lemma** [2.5] The statement

```
while b do S
```

is semantically equivalent to

```
if b then (S; while b do S) else skip.
```

---

**Proof**

Part I: (*) $\Rightarrow$ (**)  
Part II: (**) $\Rightarrow$ (*)

\[
\langle \text{while } b \text{ do } S, s \rangle \rightarrow s'' 
\]

\((*)\)

\[
\langle \text{if } b \text{ then } (S; \text{while } b \text{ do } S) \text{ else skip, } s \rangle \rightarrow s'' 
\]

\((**)\)
Because (*) holds we know that we have a derivation tree $T$ for it. It can have one of two forms depending on whether it has been constructed using the rule $\text{while}_\text{ns}^{\text{tt}}$ or the axiom $\text{while}_\text{ns}^{\text{ff}}$. In the first case the derivation tree $T$ has the form:

$$
\begin{array}{c}
T_1 & T_2 \\
\hline
\langle \text{while } b \text{ do } S, s \rangle \rightarrow s''
\end{array}
$$

where $T_1$ is a derivation tree with root $\langle S, s \rangle \rightarrow s'$ and $T_2$ is a derivation tree with root $\langle \text{while } b \text{ do } S, s' \rangle \rightarrow s''$. Furthermore, $\mathcal{B}[b]s = \text{tt}$. Using the derivation trees $T_1$ and $T_2$ as the premises for the rules $\text{comp}_\text{ns}$ we can construct the derivation tree:

$$
\begin{array}{c}
T_1 & T_2 \\
\hline
\langle S; \text{while } b \text{ do } S, s \rangle \rightarrow s''
\end{array}
$$

Using that $\mathcal{B}[b]s = \text{tt}$ we can use the rule $\text{if}_\text{ns}^{\text{tt}}$ to construct the derivation tree

$$
\begin{array}{c}
T_1 & T_2 \\
\hline
\hline
\langle S; \text{while } b \text{ do } S, s \rangle \rightarrow s''
\end{array}
$$

$$
\langle \text{if } b \text{ then } (S; \text{while } b \text{ do } S) \text{ else skip}, s \rangle \rightarrow s''
$$

thereby showing that (**) holds.
• **Summary**: Big-step operational semantics
  ✦ Models relations between syntax, states, values.
  ✦ Rule-based modeling (conclusion, premises).
  ✦ Computations are derivation trees.
  ✦ Induction proofs are a key tool in semantics.

• **Prepping**: “Semantics with applications”
  ✦ Chapter 1 and Chapter 2.1

• **Outlook**:
  ✦ Small-step semantics
  ✦ Type systems