```
x = 1
```

```
let x = 1 in ...
```

```
x(1).
```

```
!x(1)
```

```
x.set(1)
```

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
  $n \in \textsf{Num}$
- variables
  $x \in \textsf{Var}$
- arithmetic expressions
  $a \in \textsf{Aexp}$
  $a ::= n \mid x \mid a_1 + a_2 \mid a_1 * a_2 \mid a_1 - a_2$
- booleans expressions
  $b \in \textsf{Bexp}$
  $b ::= \texttt{true} \mid \texttt{false} \mid a_1 = a_2 \mid a_1 \leq a_2 \mid \neg\, b \mid b_1 \wedge b_2$
- statements
  $S \in \textsf{Stm}$
  $S ::= x := a \mid \texttt{skip} \mid S_1; S_2 \mid \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \mid \texttt{while}\ b\ \texttt{do}\ S$

This slide is derived from the book & slides by Nielson & Nielson: “Semantics with applications” (1991 & 1999+).

Semantic categories of the *While* language

Natural numbers

$\mathsf{N} = \{0, 1, 2, \cdots\}$

Truth values

$\mathsf{T} = \{\mathsf{tt}, \mathsf{ff}\}$

States

$\mathsf{State} = \mathsf{Var} \to \mathsf{N}$

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

$\mathcal{N} : \mathsf{Num} \to \mathsf{N}$

Variables

$s \in \mathsf{State} = \mathsf{Var} \to \mathsf{N}$

Arithmetic expressions

$\mathcal{A} : \mathsf{Aexp} \to (\mathsf{State} \to \mathsf{N})$

Boolean expressions

$\mathcal{B} : \mathsf{Bexp} \to (\mathsf{State} \to \mathsf{T})$

Statements

$\mathcal{S} : \mathsf{Stm} \to (\mathsf{State} \hookrightarrow \mathsf{State})$

Semantics of arithmetic expressions

$$
\begin{array}{lcl}
\mathcal{A}[n]s & = & \mathcal{N}[n] \\
\mathcal{A}[x]s & = & s\ x \\
\mathcal{A}[a_1 + a_2]s & = & \mathcal{A}[a_1]s + \mathcal{A}[a_2]s \\
\mathcal{A}[a_1\ *\ a_2]s & = & \mathcal{A}[a_1]s\ *\ \mathcal{A}[a_2]s \\
\mathcal{A}[a_1 - a_2]s & = & \mathcal{A}[a_1]s - \mathcal{A}[a_2]s
\end{array}
$$

Semantics of boolean expressions

$$
\begin{aligned}
\mathcal{B}[\![\texttt{true}]\!]s &= \mathsf{tt} \\
\mathcal{B}[\![\texttt{false}]\!]s &= \mathsf{ff} \\
\mathcal{B}[\![a_1 = a_2]\!]s &= \begin{cases} \mathsf{tt} & \text{if } \mathcal{A}[\![a_1]\!]s = \mathcal{A}[\![a_2]\!]s \\ \mathsf{ff} & \text{if } \mathcal{A}[\![a_1]\!]s \neq \mathcal{A}[\![a_2]\!]s \end{cases} \\
\mathcal{B}[\![a_1 \leq a_2]\!]s &= \begin{cases} \mathsf{tt} & \text{if } \mathcal{A}[\![a_1]\!]s \leq \mathcal{A}[\![a_2]\!]s \\ \mathsf{ff} & \text{if } \mathcal{A}[\![a_1]\!]s \not\leq \mathcal{A}[\![a_2]\!]s \end{cases} \\
\mathcal{B}[\![\neg b]\!]s &= \begin{cases} \mathsf{tt} & \text{if } \mathcal{B}[\![b]\!]s = \mathsf{ff} \\ \mathsf{ff} & \text{if } \mathcal{B}[\![b]\!]s = \mathsf{tt} \end{cases} \\
\mathcal{B}[\![b_1 \wedge b_2]\!]s &= \begin{cases} \mathsf{tt} & \text{if } \mathcal{B}[\![b_1]\!]s = \mathsf{tt} \\ & \quad \text{and } \mathcal{B}[\![b_2]\!]s = \mathsf{tt} \\ \mathsf{ff} & \text{if } \mathcal{B}[\![b_1]\!]s = \mathsf{ff} \\ & \quad \text{or } \mathcal{B}[\![b_2]\!]s = \mathsf{ff} \end{cases}
\end{aligned}
$$

Semantics of statements

$[\text{ass}_{\text{ns}}]$ $\quad \langle x := a,\ s\rangle \to s[x \mapsto \mathcal{A}[\![a]\!]s]$

$[\text{skip}_{\text{ns}}]$ $\quad \langle \texttt{skip},\ s\rangle \to s$

$$[\text{comp}_{\text{ns}}] \quad \frac{\langle S_1,\ s\rangle \to s',\ \langle S_2,\ s'\rangle \to s''}{\langle S_1;S_2,\ s\rangle \to s''}$$

$$[\text{if}^{\text{tt}}_{\text{ns}}] \quad \frac{\langle S_1,\ s\rangle \to s'}{\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2,\ s\rangle \to s'} \ \text{if}\ \mathcal{B}[\![b]\!]s = \mathbf{tt}$$

$$[\text{if}^{\text{ff}}_{\text{ns}}] \quad \frac{\langle S_2,\ s\rangle \to s'}{\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2,\ s\rangle \to s'} \ \text{if}\ \mathcal{B}[\![b]\!]s = \mathbf{ff}$$

$$[\text{while}^{\text{tt}}_{\text{ns}}] \quad \frac{\langle S,\ s\rangle \to s',\ \langle \texttt{while}\ b\ \texttt{do}\ S,\ s'\rangle \to s''}{\langle \texttt{while}\ b\ \texttt{do}\ S,\ s\rangle \to s''} \ \text{if}\ \mathcal{B}[\![b]\!]s = \mathbf{tt}$$

$[\text{while}^{\text{ff}}_{\text{ns}}]$ $\quad \langle \texttt{while}\ b\ \texttt{do}\ S,\ s\rangle \to s\ \text{if}\ \mathcal{B}[\![b]\!]s = \mathbf{ff}$

Derivation trees

$$\dfrac{\dfrac{\langle \mathtt{z{:=}x},\ s_0\rangle \to s_1 \qquad \langle \mathtt{x{:=}y},\ s_1\rangle \to s_2}{\langle \mathtt{z{:=}x;\ x{:=}y},\ s_0\rangle \to s_2} \qquad \langle \mathtt{y{:=}z},\ s_2\rangle \to s_3}{\langle \mathtt{z{:=}x;\ x{:=}y;\ y{:=}z},\ s_0\rangle \to s_3}$$

Derivation tree

$$\begin{aligned} s_0 &= [\mathtt{x}{\mapsto}\mathbf{5},\ \mathtt{y}{\mapsto}\mathbf{7},\ \mathtt{z}{\mapsto}\mathbf{0}] \\ s_1 &= [\mathtt{x}{\mapsto}\mathbf{5},\ \mathtt{y}{\mapsto}\mathbf{7},\ \mathtt{z}{\mapsto}\mathbf{5}] \\ s_2 &= [\mathtt{x}{\mapsto}\mathbf{7},\ \mathtt{y}{\mapsto}\mathbf{7},\ \mathtt{z}{\mapsto}\mathbf{5}] \\ s_3 &= [\mathtt{x}{\mapsto}\mathbf{7},\ \mathtt{y}{\mapsto}\mathbf{5},\ \mathtt{z}{\mapsto}\mathbf{5}] \end{aligned}$$

States

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

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

% Subtraction
...
% Multiplication
...
```

Boolean expression evaluation

```
evalb(true,_,tt).
evalb(false,_,ff).

evalb(not(B),M,V) :-
 evalb(B,M,V1),
 not(V1,V).

evalb(and(B1,B2),M,V) :-
 evalb(B1,M,V1),
 evalb(B2,M,V2),
 and(V1,V2,V).
...
```

Skip statement

```
exec(skip,M,M).
```

Sequential composition

```
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,_)|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

$$
\begin{array}{lcl}
S & ::= & x := a \mid \texttt{skip} \mid S_1 \texttt{ ; } S_2 \\
 & \mid & \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2 \\
 & \mid & \texttt{while } b \texttt{ do } S \\
 & \mid & \texttt{begin } D_V \; D_P \; S \texttt{ end} \\
 & \mid & \texttt{call } p \\
D_V & ::= & \texttt{var } x := a ; D_V \mid \varepsilon \\
D_P & ::= & \texttt{proc } p \texttt{ is } S ; D_P \mid \varepsilon
\end{array}
$$

Semantics of var declarations

Extension of semantics of statements:

$$\frac{(D_V,\, s) \rightarrow_D s',\, (S,\, s') \rightarrow s''}{(\texttt{begin}\ D_V\ S\ \texttt{end},\, s) \rightarrow s''[\mathrm{DV}(D_V) \longmapsto s]}$$

Semantics of variable declarations:

$$\frac{(D_V,\, s[x \mapsto \mathcal{A}[a]s]) \rightarrow_D s'}{(\texttt{var}\ x := a;\, D_V,\, s) \rightarrow_D s'}$$

$$(\varepsilon,\, s) \rightarrow_D s$$

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

```
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
```

- 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

[ass$_{ns}$] $env_P \vdash \langle x := a,\, s\rangle \to s[x \mapsto \mathcal{A}[\![a]\!]s]$

[skip$_{ns}$] $env_P \vdash \langle \texttt{skip},\, s\rangle \to s$

[comp$_{ns}$] $$\frac{env_P \vdash \langle S_1,\, s\rangle \to s',\; env_P \vdash \langle S_2,\, s'\rangle \to s''}{env_P \vdash \langle S_1;S_2,\, s\rangle \to s''}$$

[if$^{tt}_{ns}$] $$\frac{env_P \vdash \langle S_1,\, s\rangle \to s'}{env_P \vdash \langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2,\, s\rangle \to s'}$$
if $\mathcal{B}[\![b]\!]s = \mathbf{tt}$

[if$^{ff}_{ns}$] $$\frac{env_P \vdash \langle S_2,\, s\rangle \to s'}{env_P \vdash \langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2,\, s\rangle \to s'}$$
if $\mathcal{B}[\![b]\!]s = \mathbf{ff}$

[while$^{tt}_{ns}$] $$\frac{env_P \vdash \langle S,\, s\rangle \to s',\; env_P \vdash \langle \texttt{while}\ b\ \texttt{do}\ S,\, s'\rangle \to s''}{env_P \vdash \langle \texttt{while}\ b\ \texttt{do}\ S,\, s\rangle \to s''}$$
if $\mathcal{B}[\![b]\!]s = \mathbf{tt}$

[while$^{ff}_{ns}$] $env_P \vdash \langle \texttt{while}\ b\ \texttt{do}\ S,\, s\rangle \to s$
if $\mathcal{B}[\![b]\!]s = \mathbf{ff}$

[block$_{ns}$] $$\frac{\langle D_V,\, s\rangle \to_D s',\; \mathrm{upd_P}(D_P,\, env_P) \vdash \langle S,\, s'\rangle \to s''}{env_P \vdash \langle \texttt{begin}\ D_V\ D_P\ S\ \texttt{end},\, s\rangle \to s''[\mathrm{DV}(D_V) \longmapsto s]}$$

[call$^{rec}_{ns}$] $$\frac{env_P \vdash \langle S,\, s\rangle \to s'}{env_P \vdash \langle \texttt{call}\ p,\, s\rangle \to s'} \quad \text{where } env_P\ p = S$$

$$\mathrm{upd_P}(\texttt{proc}\ p\ \texttt{is}\ S;\ D_P,\, env_P) = \mathrm{upd_P}(D_P,\, env_P[p \mapsto S])$$

$$\mathrm{upd_P}(\varepsilon,\, env_P) = env_P$$

NS with dynamic scope rules using an environment

$$\mathbf{Env_P} = \mathbf{Pname} \hookrightarrow \mathbf{Stm}$$

Dynamic scope for variables
Static scope for 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
```

- 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

Dynamic scope for variables Static scope for procedures

- Updated environment

$$\mathbf{Env}_{\mathrm{P}} = \mathbf{Pname} \hookrightarrow \mathbf{Stm} \times \mathbf{Env}_{\mathrm{P}}$$

- Updated environment update

$$\mathrm{upd}_{\mathrm{P}}(\texttt{proc}\ p\ \texttt{is}\ S;\ D_P,\ env_P) = \mathrm{upd}_{\mathrm{P}}(D_P,\ env_P[p\mapsto(S,\ env_P)])$$

$$\mathrm{upd}_{\mathrm{P}}(\varepsilon,\ env_P) = env_P$$

- Updated rule for calls

$$\frac{env'_P \vdash \langle S,\ s\rangle \to s'}{env_P \vdash \langle \texttt{call}\ p,\ s\rangle \to s'}$$

$$\text{where } env_P\ p = (S,\ env'_P)$$

- Recursive calls

$$\frac{env'_P[p\mapsto(S,\ env'_P)] \vdash \langle S,\ s\rangle \to s'}{env_P \vdash \langle \texttt{call}\ p,\ s\rangle \to s'}$$

$$\text{where } env_P\ p = (S,\ env'_P)$$

Static scope for variables and 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
```

- 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

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 $x \in \mathsf{FV}(a)$. Then

$$\mathcal{A}[a]s = \mathcal{A}[a]s'$$

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

Free variables in arithmetic expressions

$$
\begin{array}{lcl}
\mathsf{FV}(n) & = & \emptyset \\
\mathsf{FV}(x) & = & \{\ x\ \} \\
\mathsf{FV}(a_1 + a_2) & = & \mathsf{FV}(a_1)\ \cup\ \mathsf{FV}(a_2) \\
\mathsf{FV}(a_1 * a_2) & = & \mathsf{FV}(a_1)\ \cup\ \mathsf{FV}(a_2) \\
\mathsf{FV}(a_1 - a_2) & = & \mathsf{FV}(a_1)\ \cup\ \mathsf{FV}(a_2)
\end{array}
$$

Proof by structural induction on the arithmetic expressions

Consider again the semantics of arithmetic expressions

$$
\begin{aligned}
\mathcal{A}[n]s &= \mathcal{N}[n] \\
\mathcal{A}[x]s &= s\ x \\
\mathcal{A}[a_1 + a_2]s &= \mathcal{A}[a_1]s + \mathcal{A}[a_2]s \\
\mathcal{A}[a_1\ *\ a_2]s &= \mathcal{A}[a_1]s\ *\ \mathcal{A}[a_2]s \\
\mathcal{A}[a_1 - a_2]s &= \mathcal{A}[a_1]s - \mathcal{A}[a_2]s
\end{aligned}
$$

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

Compositional Definitions

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

2: The semantics is defined by *compositional* definitions of a function: There is a *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.

Structural Induction

1: Prove that the property holds for all the *basis* elements of the syntactic category.

2: Prove that the property holds for all the *composite* elements of the syntactic category: Assume that the property holds for all the immediate constituents of the element (this is called the *induction hypothesis*) and prove that it also holds for the element itself.

Let s and s' be two states satisfying

$$s\ x = s'\ x$$

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

$$\mathcal{A}[a]s = \mathcal{A}[a]s'$$

$$\begin{array}{lcl}
\mathcal{A}[\![n]\!]s & = & \mathcal{N}[\![n]\!] \\
\mathcal{A}[\![x]\!]s & = & s\ x \\
\mathcal{A}[\![a_1 + a_2]\!]s & = & \mathcal{A}[\![a_1]\!]s + \mathcal{A}[\![a_2]\!]s \\
\mathcal{A}[\![a_1 \star a_2]\!]s & = & \mathcal{A}[\![a_1]\!]s \star \mathcal{A}[\![a_2]\!]s \\
\mathcal{A}[\![a_1 - a_2]\!]s & = & \mathcal{A}[\![a_1]\!]s - \mathcal{A}[\![a_2]\!]s
\end{array}$$

Table 1.1: The semantics of arithmetic expressions

Proofs for basis elements

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

The case x: From Table 1.1 we have $\mathcal{A}[\![x]\!]s = s\ x$ as well as $\mathcal{A}[\![x]\!]s' = s'\ x$. From the assumptions of the lemma we get $s\ x = s'\ x$ because $x \in \mathrm{FV}(x)$ 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 $x \in \mathsf{FV}(a)$. Then

$$\mathcal{A}[a]s = \mathcal{A}[a]s'$$

$\mathcal{A}[\![n]\!]s$	$=$	$\mathcal{N}[\![n]\!]$
$\mathcal{A}[\![x]\!]s$	$=$	$s\ x$
$\mathcal{A}[\![a_1 + a_2]\!]s$	$=$	$\mathcal{A}[\![a_1]\!]s + \mathcal{A}[\![a_2]\!]s$
$\mathcal{A}[\![a_1 \star a_2]\!]s$	$=$	$\mathcal{A}[\![a_1]\!]s \star \mathcal{A}[\![a_2]\!]s$
$\mathcal{A}[\![a_1 - a_2]\!]s$	$=$	$\mathcal{A}[\![a_1]\!]s - \mathcal{A}[\![a_2]\!]s$

Table 1.1: The semantics of arithmetic expressions

Proofs for composite elements

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

The cases $a_1 - a_2$ and $a_1 \star a_2$ follow the same pattern and are omitted.

Proof by structural induction on the arithmetic expressions

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' and s''

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

Proof We assume $(S, s) \to s'$.
We prove that if $(S, s) \to s''$ then $s' = s''$.

We proceed by induction on the inference of $(S, s) \to s'$.

Proof by induction on the shape of derivation trees

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.

Structural induction on syntactical categories is ***not*** applicable because of the non-compositional semantics of while!

$$[\text{while}_{\text{ns}}^{\text{tt}}] \qquad \frac{\langle S,\, s\rangle \to s',\ \langle \texttt{while}\ b\ \texttt{do}\ S,\, s'\rangle \to s''}{\langle \texttt{while}\ b\ \texttt{do}\ S,\, s\rangle \to s''} \ \text{ if } \mathcal{B}[\![b]\!]s = \mathbf{tt}$$

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.

Theorem [2.9]

The natural semantics of While is deterministic, that is for all statements S of While and all states s, s' and s''
if $(S, s) \to s'$ and $(S, s) \to s''$
then $s' = s''$.

Proof: We assume that $\langle S,\ s\rangle \to s'$ and shall prove that

if $\langle S,\ s\rangle \to s''$ then $s' = s''$.

We shall proceed by induction on the shape of the derivation tree for $\langle S,\ s\rangle \to s'$.

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

The case $[\text{skip}_{\text{ns}}]$: Analogous.

Proof by induction on the shape of derivation trees

The case [comp$_{\mathrm{ns}}$]: Assume that

$$\langle S_1;S_2,\ s\rangle \to s'$$

holds because

$$\langle S_1,\ s\rangle \to s_0 \text{ and } \langle S_2,\ s_0\rangle \to s'$$

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

$$\langle S_1,\ s\rangle \to s_1 \text{ and } \langle S_2,\ s_1\rangle \to s''$$

The induction hypothesis can be applied to the premise $\langle S_1,\ s\rangle \to s_0$ and from $\langle S_1,\ s\rangle \to s_1$ we get $s_0 = s_1$. Similarly, the induction hypothesis can be applied to the premise $\langle S_2,\ s_0\rangle \to s'$ and from $\langle S_2,\ s_0\rangle \to s''$ we get $s' = s''$ as required.

Proof by induction on the shape of derivation trees

The case $[\text{if}_{\text{ns}}^{\text{tt}}]$: Assume that

$$\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2,\ s\rangle \to s'$$

holds because

$$\mathcal{B}[\![b]\!]s = \mathbf{tt} \text{ and } \langle S_1,\ s\rangle{\to}s'$$

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

$$\langle S_1,\ s\rangle \to s''$$

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

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

Proof by induction on the shape of derivation trees

Non-compositional semantics is Ok for this proof scheme.

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

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

Proof by induction on the shape of derivation trees

Yet another property of the semantics

Lemma [2.5] The statement

$$\texttt{while}\ b\ \texttt{do}\ S$$

is semantically equivalent to

$$\texttt{if}\ b\ \texttt{then}\ (S;\ \texttt{while}\ b\ \texttt{do}\ S)\ \texttt{else skip}.$$

Proof

Part I: (*) ⇒ (**)

Part II: (**) ⇒ (*)

$$\langle \texttt{while}\ b\ \texttt{do}\ S,\ s\rangle \to s'' \qquad (*)$$

$$\langle \texttt{if}\ b\ \texttt{then}\ (S;\ \texttt{while}\ b\ \texttt{do}\ S)\ \texttt{else skip},\ s\rangle \to s'' \qquad (**)$$

Proof

only (*) ⇒ (**)

only for **tt**

No induction needed here.

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 [while$^{\text{tt}}_{\text{ns}}$] or the axiom [while$^{\text{ff}}_{\text{ns}}$]. In the first case the derivation tree T has the form:

$$\frac{T_1 \qquad T_2}{\langle \texttt{while}\ b\ \texttt{do}\ S,\ s\rangle \to s''}$$

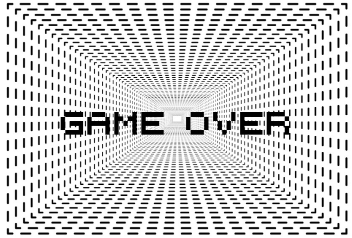
where T_1 is a derivation tree with root $\langle S,\ s\rangle \to s'$ and T_2 is a derivation tree with root $\langle \texttt{while}\ b\ \texttt{do}\ S,\ s'\rangle \to s''$. Furthermore, $\mathcal{B}[\![b]\!]s = \mathbf{tt}$. Using the derivation trees T_1 and T_2 as the premises for the rules [comp$_{\text{ns}}$]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

$$\frac{T_1 \qquad T_2}{\langle S;\ \texttt{while}\ b\ \texttt{do}\ S,\ s\rangle \to s''}$$

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

$$\frac{\dfrac{T_1 \qquad\qquad\qquad T_2}{\langle S;\ \texttt{while}\ b\ \texttt{do}\ S,\ s\rangle \to s''}}{\langle \texttt{if}\ b\ \texttt{then}\ (S;\ \texttt{while}\ b\ \texttt{do}\ S)\ \texttt{else}\ \texttt{skip},\ s\rangle \to s''}$$

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*