Programming Language Theory

Operational Semantics
(Summer Semester 2013/14)

Ralf Lämmel
“Semantics with applications”

Chapter 1
Chapter 2.1, 2.2, 2.5
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
  \( a \in \text{Aexp} \)
  \[
  a ::= n \mid x \mid a_1 + a_2 \\
  \mid a_1 \cdot a_2 \mid a_1 - a_2
  \]

- 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 **While** language

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

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

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

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
\[ N : \text{Num} \to N \]

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

Arithmetic expressions
\[ A : \text{Aexp} \to (\text{State} \to N) \]

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

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

We do not define all these functions “directly”. Especially the last one is defined as a relation based on rules (with premises and conclusions).
Semantics of arithmetic expressions

\[ A[n]s = N[n] \]
\[ A[x]s = s x \]
\[ A[a_1 + a_2]s = A[a_1]s + A[a_2]s \]
\[ A[a_1 \ast a_2]s = A[a_1]s \ast A[a_2]s \]
\[ A[a_1 - a_2]s = A[a_1]s - A[a_2]s \]

Semantics of boolean expressions

\[ B[\text{true}]s = \text{tt} \]
\[ B[\text{false}]s = \text{ff} \]
\[ B[a_1 = a_2]s = \begin{cases} 
\text{tt} & \text{if } A[a_1]s = A[a_2]s \\
\text{ff} & \text{if } A[a_1]s \neq A[a_2]s 
\end{cases} \]
\[ B[a_1 \leq a_2]s = \begin{cases} 
\text{tt} & \text{if } A[a_1]s \leq A[a_2]s \\
\text{ff} & \text{if } A[a_1]s \nleq A[a_2]s 
\end{cases} \]
\[ B[\neg b]s = \begin{cases} 
\text{tt} & \text{if } B[b]s = \text{ff} \\
\text{ff} & \text{if } B[b]s = \text{tt} 
\end{cases} \]
\[ B[b_1 \land b_2]s = \begin{cases} 
\text{tt} & \text{if } B[b_1]s = \text{tt} \\
& \text{and } B[b_2]s = \text{tt} \\
\text{ff} & \text{if } B[b_1]s = \text{ff} \\
& \text{or } B[b_2]s = \text{ff} 
\end{cases} \]
Semantics of statements

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

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

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

\[\text{[if}^\tt_{ns}] \quad \frac{\langle S_1, s \rangle \rightarrow s'}{\langle \text{if } b \text{ then } S_1 \text{ else } S_2, s \rangle \rightarrow s'} \quad \text{if } B[b]s = \tt\]

\[\text{[if}^\ff_{ns}] \quad \frac{\langle S_2, s \rangle \rightarrow s'}{\langle \text{if } b \text{ then } S_1 \text{ else } S_2, s \rangle \rightarrow s'} \quad \text{if } B[b]s = \ff\]

\[\text{[while}^\tt_{ns}] \quad \frac{\langle S, s \rangle \rightarrow s', \langle \text{while } b \text{ do } S, s' \rangle \rightarrow s''}{\langle \text{while } b \text{ do } S, s \rangle \rightarrow s''} \quad \text{if } B[b]s = \tt\]

\[\text{[while}^\ff_{ns}] \quad \langle \text{while } b \text{ do } S, s \rangle \rightarrow s \quad \text{if } B[b]s = \ff\]
Derivation trees

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

\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*}
Blocks and procedures

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

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

Extension of semantics of statements:

\[
(D_V, s) \xrightarrow{D} s', (S, s') \xrightarrow{} s''
\]
\[
(\text{begin } D_V S \text{ end}, s) \xrightarrow{} s''[DV(D_V) \xrightarrow{} s]
\]

Semantics of variable declarations:

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

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

```plaintext
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 \( y = 6 \)

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

\[
\text{Env}_P = \text{Pname} \leftrightarrow \text{Stm}
\]

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

\[
\text{upd}_P(\varepsilon, \text{env}_P) = \text{env}_P
\]

© Ralf Lämmel, 2009-2012 unless noted otherwise
Dynamic scope for variables
Static scope for procedures

• Execution
  ✦ call q
  ✦ call p (calls outer, say global p)
  ✦ $x := x \times 2$ (affects inner, say local $x$)
  ✦ $y := x$ (obviously accesses local $x$)
• Final value of $y = 10$

```plaintext
begin
  var x := 0;
  proc p is x := x \times 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
\[ \text{env}_P' \vdash \langle S, s \rangle \rightarrow s' \]
\[ \text{env}_P \vdash \langle \text{call } p, s \rangle \rightarrow s' \]
where \( \text{env}_P p = (S, \text{env}_P') \)

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


Formal semantics omitted here.
Small-step Operational Semantics
(aka Structured Operational Semantics)

Ralf Lämmel
Big-step style

\[
\begin{align*}
\text{[comp}_{\text{ns}}] & \quad \frac{\langle S_1, s \rangle \rightarrow s', \langle S_2, s' \rangle \rightarrow s''}{\langle S_1; S_2, s \rangle \rightarrow s''}
\end{align*}
\]

Easier to understand

Small-step style

\[
\begin{align*}
\text{[comp}^1_{\text{sos}}] & \quad \frac{\langle S_1, s \rangle \Rightarrow \langle S'_1, s' \rangle}{\langle S_1; S_2, s \rangle \Rightarrow \langle S'_1; S_2, s' \rangle} \\
\text{[comp}^2_{\text{sos}}] & \quad \frac{\langle S_1, s \rangle \Rightarrow s'}{\langle S_1; S_2, s \rangle \Rightarrow \langle S_2, s' \rangle}
\end{align*}
\]

“More versatile”
SOS (statements)

\[
\begin{align*}
[\text{ass}_\text{sos}] & \quad \langle x := a, s \rangle \Rightarrow s[x \mapsto A[a]]s \\
[\text{skip}_\text{sos}] & \quad \langle \text{skip}, s \rangle \Rightarrow s \\
[\text{comp}^1_\text{sos}] & \quad \frac{\langle S_1, s \rangle \Rightarrow \langle S'_1, s' \rangle}{\langle S_1; S_2, s \rangle \Rightarrow \langle S'_1; S_2, s' \rangle} \\
[\text{comp}^2_\text{sos}] & \quad \frac{\langle S_1, s \rangle \Rightarrow s'}{\langle S_1; S_2, s \rangle \Rightarrow \langle S_2, s' \rangle} \\
[\text{if}^{tt}_\text{sos}] & \quad \langle \text{if } b \text{ then } S_1 \text{ else } S_2, s \rangle \Rightarrow \langle S_1, s \rangle \text{ if } B[b]s = \text{tt} \\
[\text{if}^{ff}_\text{sos}] & \quad \langle \text{if } b \text{ then } S_1 \text{ else } S_2, s \rangle \Rightarrow \langle S_2, s \rangle \text{ if } B[b]s = \text{ff} \\
[\text{while}_\text{sos}] & \quad \langle \text{while } b \text{ do } S, s \rangle \Rightarrow \\
& \quad \quad \langle \text{if } b \text{ then } (S; \text{while } b \text{ do } S) \text{ else skip}, s \rangle
\end{align*}
\]
Transition systems in semantics
Transition systems

\((\Gamma, T, \triangleright)\)

- \(\Gamma\): a set of configurations
- \(T\): a set of terminal configurations
  \(T \subseteq \Gamma\)
- \(\triangleright\): a transition relation
  \(\triangleright \subseteq \Gamma \times \Gamma\)
Big step versus small step

- Big-step semantics
  - aka Natural semantics
  - **One** (fewer) transition(s)
  - Computation steps modeled by derivation tree

- Small-step semantics
  - aka Structured Operational Semantics (SOS)
  - **Many** transitions
  - Computation steps modeled by transitions
Big step operational semantics: describe how the “final” result of the computation is obtained.

Transition system: \((\Gamma, T, \rightarrow)\)

- \(\Gamma = \{(S, s) \mid S \in \text{While}, s \in \text{State}\} \cup \text{State}\)
- \(T = \text{State}\)
- \(\rightarrow \subseteq \{(S, s) \mid S \in \text{While}, s \in \text{State}\} \times \text{State}\)

Typical transition:

\((S, s) \rightarrow s'\)

where

- \(S\) is the program
- \(s\) is the initial state
- \(s'\) is the final state
Small step operational semantics: describe how the individual steps of the computation take place.

Transition system: $\langle \Gamma, T, \Rightarrow \rangle$

- $\Gamma = \{(S, s) \mid S \in \text{While}, s \in \text{State}\} \cup \text{State}$
- $T = \text{State}$
- $\Rightarrow \subseteq \{(S, s) \mid S \in \text{While}, s \in \text{State}\} \times \Gamma$

Two typical transitions:

- the computation has not been completed after one step of computation:
  $$(S, s) \Rightarrow (S', s')$$

- the computation is completed after one step of computation:
  $$(S, s) \Rightarrow s'$$

Big step versus small step

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

Derivation tree

Transition = big step

\[
\begin{align*}
 s_0 & = [x \mapsto 5, y \mapsto 7, z \mapsto 0] \\
 s_1 & = [x \mapsto 5, y \mapsto 7, z \mapsto 5] \\
 s_2 & = [x \mapsto 7, y \mapsto 7, z \mapsto 5] \\
 s_3 & = [x \mapsto 7, y \mapsto 5, z \mapsto 5]
\end{align*}
\]
Big step versus **small step**

A “small” derivation tree for each step

\[
\begin{align*}
\langle z := x, s_0 \rangle &\Rightarrow s_0[z \mapsto 5] \\
\langle z := x; x := y, s_0 \rangle &\Rightarrow \langle x := y, s_0[z \mapsto 5] \rangle \\
\langle (z := x; x := y); y := z, s_0 \rangle &\Rightarrow \langle x := y; y := z, s_0[z \mapsto 5] \rangle
\end{align*}
\]
Big step versus **small step**

Program configuration

```plaintext
\langle z := x; x := y; y := z, \rangle \Rightarrow \langle x := y; y := z, \rangle \Rightarrow \langle y := z, \rangle \Rightarrow \langle x := 5, y := 7, z := 0 \rangle
```

Variable assignments (states)

Final state

Transition = **small** step

Many steps = derivation sequence

Execution of \( \langle S, s \rangle \) terminates successfully if \( \langle S, s \rangle \Rightarrow^k s' \) for some \( k \) and \( s' \).

Execution loops if there is an infinite derivation sequence.
Define functions for meanings in terms of the transition relations

\[ S_{ns}: \text{Stm} \rightarrow (\text{State} \leftrightarrow \text{State}) \]

\[ S_{sos}: \text{Stm} \rightarrow (\text{State} \leftrightarrow \text{State}) \]

\[ S_{ns}[S]_s = \begin{cases} 
    s' & \text{if } \langle S, s \rangle \rightarrow s' \\
    \text{undef} & \text{otherwise}
\end{cases} \]

\[ S_{sos}[S]_s = \begin{cases} 
    s' & \text{if } \langle S, s \rangle \Rightarrow^* s' \\
    \text{undef} & \text{otherwise}
\end{cases} \]
Extensions of **While**

\[
S ::= x := a \mid \text{skip} \mid S_1; S_2 \\
    | \text{if } b \text{ then } S_1 \text{ else } S_2 \\
    | \text{while } b \text{ do } S \\
    | \text{abort} \\
    | S_1 \text{ or } S_2 \\
    | S_1 \text{ par } S_2
\]
Adding abortion

Configurations:

\[ \{(S, s) \mid S \in \text{While}^{abort}, s \in \text{State}\} \cup \text{State} \]

Transition relation for NS:
unchanged

Transition relation for SOS:
unchanged
Understanding the characteristics of small- and big-step style

- Informal semantics
  - **skip**: The program should terminate normally.
  - **abort**: The program should terminate abnormally.
  - **while true do skip**: The program should not terminate.
Understanding the characteristics of small- and big-step style

- **Small**-step style
  - skip: \( \langle \text{skip}, s \rangle \Rightarrow s \)
  - abort: \( \langle \text{abort}, s \rangle \)
  - while true do skip: \( \langle \text{while true do skip}, s \rangle \)
    \[ \Rightarrow \langle \text{if true then (skip; while true do skip) else skip}, s \rangle \]
    \[ \Rightarrow \langle \text{skip; while true do skip}, s \rangle \]
    \[ \Rightarrow \langle \text{while true do skip}, s \rangle \]
    \[ \Rightarrow \cdots \]
Understanding the characteristics of small- and big-step style

- **Big**-step style
  - skip: \(\langle \text{skip}, s \rangle \rightarrow s\)
  - abort: No transition
  - while true do skip: No transition.
Adding nondeterminism

\[ x := 1 \text{ or } (x := 2; x := x + 2) \text{ assigns } 1 \text{ or } 4 \text{ to } x. \]

\[
\begin{align*}
\text{or}^1_{\text{sos}} & : \quad \langle S_1 \text{ or } S_2, s \rangle \Rightarrow \langle S_1, s \rangle \\
\text{or}^2_{\text{sos}} & : \quad \langle S_1 \text{ or } S_2, s \rangle \Rightarrow \langle S_2, s \rangle \\
\text{or}^1_{\text{ns}} & : \quad \frac{\langle S_1, s \rangle \rightarrow s'}{\langle S_1 \text{ or } S_2, s \rangle \rightarrow s'} \\
\text{or}^2_{\text{ns}} & : \quad \frac{\langle S_2, s \rangle \rightarrow s'}{\langle S_1 \text{ or } S_2, s \rangle \rightarrow s'}
\end{align*}
\]
NS vs. SOS

Consider the following two programs:
(while true do skip) or (x := 2; x := x+2)

- **Natural semantics**: There is one derivation tree for the terminating choice. Thus, the semantics “suppresses” looping.

- **SOS**: There is one finite and another infinite derivation sequence. Thus, the semantics makes non-termination “observable”.
Adding parallelism

\[ x := 1 \text{ par } (x := 2; x := x+2) \] assigns 1, 3, or 4 to \( x \).

Transition relation for SOS:

\[
\frac{(S_1, s) \Rightarrow (S'_1, s')}{(S_1 \text{ par } S_2, s) \Rightarrow (S'_1 \text{ par } S_2, s')}
\]

\[
\frac{(S_2, s) \Rightarrow s'}{(S_1 \text{ par } S_2, s) \Rightarrow (S_2, s')}
\]

\[
\frac{(S_2, s) \Rightarrow (S'_2, s')}{(S_1 \text{ par } S_2, s) \Rightarrow (S_1 \text{ par } S'_2, s')}
\]

\[
\frac{(S_2, s) \Rightarrow s'}{(S_1 \text{ par } S_2, s) \Rightarrow (S_1, s')}
\]

Transition relation for NS:

\[
\frac{(S_1, s) \rightarrow s', (S_2, s') \rightarrow s''}{(S_1 \text{ par } S_2, s) \rightarrow s''}
\]

\[
\frac{(S_2, s) \rightarrow s', (S_1, s') \rightarrow s''}{(S_1 \text{ par } S_2, s) \rightarrow s''}
\]

\[
\frac{(S_2, s) \rightarrow s'}{(S_1 \text{ par } S_2, s) \rightarrow (S_1, s')}
\]
NS vs. SOS

- **Natural semantics**: Each constituent of par is executed in one big step. Hence, interleaving of computations is not achieved.
  
  \[
  x := 1 \text{ par } (x := 2; x := x+2) \text{ evaluates to } 1, 4.
  \]

- **SOS**: The constituents of par are executed in many small steps. Hence interleaving of computations is achieved.
  
  \[
  x := 1 \text{ par } (x := 2; x := x+2) \text{ evaluates to } 1, 3, \text{ or } 4.
  \]
What sort of reasoning are we interested in?

- Is a given semantics deterministic or total?
- Are two given semantics equivalent?
- Is a given program transformation semantically correct?
- ...

Styles of semantics and proofs

✦ Three approaches to semantics:
  ★ Compositional definitions
  ★ Natural semantics
  ★ SOS

✦ Three corresponding proof principles:
  ★ Induction on the syntactic structure
  ★ Induction on the shape of derivation trees
  ★ Induction on the length of derivation sequences

Of course, we may also use other proof techniques.

A property of the expression 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.

Free variables in arithmetic expressions

\[
\begin{align*}
\text{FV}(n) & = \emptyset \\
\text{FV}(x) & = \{ x \} \\
\text{FV}(a_1 + a_2) & = \text{FV}(a_1) \cup \text{FV}(a_2) \\
\text{FV}(a_1 \ast a_2) & = \text{FV}(a_1) \cup \text{FV}(a_2) \\
\text{FV}(a_1 - a_2) & = \text{FV}(a_1) \cup \text{FV}(a_2)
\end{align*}
\]

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

\[ A[n]s = N[n] \]
\[ A[x]s = s \times \]
\[ A[a_1 + a_2]s = A[a_1]s + A[a_2]s \]
\[ A[a_1 \times a_2]s = A[a_1]s \times A[a_2]s \]
\[ A[a_1 - a_2]s = A[a_1]s - A[a_2]s \]

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

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td>The syntactic category is specified by an abstract syntax giving the <em>basis elements</em> and the <em>composite elements</em>. The composite elements have a unique decomposition into their immediate constituents.</td>
</tr>
<tr>
<td>2:</td>
<td>The semantics is defined by <em>compositional</em> definitions of a function: There is a <em>semantic clause</em> 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.</td>
</tr>
</tbody>
</table>
### 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.
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.
Let $s$ and $s'$ be two states satisfying
\[ s \cdot x = s' \cdot x \]
for all $x \in \text{FV}(a)$. Then
\[ \mathcal{A}[a]s = \mathcal{A}[a]s' \]

<table>
<thead>
<tr>
<th>$\mathcal{A}[n]s$</th>
<th>$= \mathcal{N}[n]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mathcal{A}[x]s$</td>
<td>$= s \cdot x$</td>
</tr>
<tr>
<td>$\mathcal{A}[a_1 + a_2]s$</td>
<td>$= \mathcal{A}[a_1]s + \mathcal{A}[a_2]s$</td>
</tr>
<tr>
<td>$\mathcal{A}[a_1 \cdot a_2]s$</td>
<td>$= \mathcal{A}[a_1]s \cdot \mathcal{A}[a_2]s$</td>
</tr>
<tr>
<td>$\mathcal{A}[a_1 - a_2]s$</td>
<td>$= \mathcal{A}[a_1]s - \mathcal{A}[a_2]s$</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 $\mathcal{A}[a_1 + a_2]s = \mathcal{A}[a_1]s + \mathcal{A}[a_2]s$ 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 $\text{FV}(a_i) \subseteq \text{FV}(a_1 + a_2)$ we can apply the induction hypothesis (that is the lemma) to $a_i$ and get $\mathcal{A}[a_i]s = \mathcal{A}[a_i]s'$. It is now easy to see that the lemma holds for $a_1 + a_2$ as well.

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

Proof by structural induction on the arithmetic expressions
A property of the statement 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 $(S, s) \rightarrow s'$ 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 $(S, s) \rightarrow s'$. 

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!

\[
[\text{while}_{\text{ns}}^{\text{tt}}] \quad \frac{\langle S, s \rangle \rightarrow s', \langle \text{while } b \text{ do } S, s' \rangle \rightarrow s''}{\langle \text{while } b \text{ do } S, s \rangle \rightarrow s''} \quad \text{if } B[b]^s = \text{tt}
\]
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 $(S, s) \rightarrow s'$ 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 [ass ns]: Then $S$ is $x := a$ and $s'$ is $s[x \rightarrow A[a]] s$. The only axiom or rule that could be used to give $⟨x := a, s⟩ \rightarrow s''$ is [ass ns] so it follows that $s''$ must be $s[x \rightarrow A[a]] s$ and thereby $s' = s''$.

The case [skip ns]: Analogous.

Proof by induction on the shape of derivation trees
The case \([\text{comp}_{ns}]\): 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 \([\text{comp}_{ns}]\) 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 [if\textsuperscript{tt}]\textsubscript{ns}: Assume that

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

holds because

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

From \(B[b]s = \text{tt}\) 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 [if\textsuperscript{tt}]\textsubscript{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 [if\textsuperscript{ff}]\textsubscript{ns}: Analogous.
Non-compositional semantics is Ok for this proof scheme.

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

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

Proof by induction on the shape of derivation trees
A property on program equivalence

**Lemma** [2.5] The statement

\[
\text{while } b \text{ do } S
\]

is semantically equivalent to

\[
\text{if } b \text{ then } (S; \text{while } b \text{ do } S) \text{ else skip.}
\]

---

**Proof**

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

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

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

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

© Ralf Lämmel, 2009-2012 unless noted otherwise
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 \([\text{while}_{\text{ns}}]\) or the axiom \([\text{while}_{\text{ns}}]\). In the first case the derivation tree \( T \) has the form:

\[
\begin{array}{c}
T_1 \\
\hline
T_2
\end{array}
\]

\( \langle \text{while } b \text{ do } S, s \rangle \rightarrow s'' \)

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, \( B[b]s = \text{tt} \). 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:

\[
\begin{array}{c}
T_1 \\
\hline
T_2
\end{array}
\]

\( \langle S; \text{while } b \text{ do } S, s \rangle \rightarrow s'' \)

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

\[
\begin{array}{c}
T_1 \\
\hline
T_2
\end{array}
\]

\( \langle S; \text{while } b \text{ do } S, s \rangle \rightarrow s'' \)

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

thereby showing that (***) holds.
The property of semantics equivalence

\[ S_{\text{ns}} : \text{Stm} \rightarrow (\text{State} \leftrightarrow \text{State}) \]

\[ S_{\text{sos}} : \text{Stm} \rightarrow (\text{State} \leftrightarrow \text{State}) \]

\[ S_{\text{ns}}[S]s = \begin{cases} 
  s' & \text{if } \langle S, s \rangle \rightarrow s' \\
  \text{undef} & \text{otherwise}
\end{cases} \]

\[ S_{\text{sos}}[S]s = \begin{cases} 
  s' & \text{if } \langle S, s \rangle \Rightarrow^* s' \\
  \text{undef} & \text{otherwise}
\end{cases} \]

**Theorem 2.26** For every statement \( S \) of While we have \( S_{\text{ns}}[S] = S_{\text{sos}}[S] \).

Proof by induction on derivation sequences
Induction on the Length of Derivation Sequences

1: Prove that the property holds for all derivation sequences of length 0.
2: Prove that the property holds for all other derivation sequences: Assume that the property holds for all derivation sequences of length at most k (this is called the *induction hypothesis*) and show that it holds for derivation sequences of length k+1.
Theorem 2.26  For every statement $S$ of While we have $S_{ns}[S] = S_{sos}[S]$.

**Proof Summary for While:**

**Equivalence of two Operational Semantics**

1:  Prove by *induction on the shape of derivation trees* that for each derivation tree in the natural semantics there is a corresponding finite derivation sequence in the structural operational semantics.

2:  Prove by *induction on the length of derivation sequences* that for each finite derivation sequence in the structural operational semantics there is a corresponding derivation tree in the natural semantics.
Theorem 2.26 For every statement \( S \) of While we have \( S_{ns}[S] = S_{sos}[S] \).

Lemma 2.27 For every statement \( S \) of While and states \( s \) and \( s' \) we have

\[
\langle S, s \rangle \rightarrow s' \text{ implies } \langle S, s \rangle \Rightarrow^* s'.
\]

So if the execution of \( S \) from \( s \) terminates in the natural semantics then it will terminate in the same state in the structural operational semantics.

Lemma 2.28 For every statement \( S \) of While, states \( s \) and \( s' \) and natural number \( k \) we have that

\[
\langle S, s \rangle \Rightarrow^k s' \text{ implies } \langle S, s \rangle \rightarrow s'.
\]

So if the execution of \( S \) from \( s \) terminates in the structural operational semantics then it will terminate in the same state in the natural semantics.

Let's focus on this lemma for the sake of exercising induction on length of derivation sequences.
\[ \langle S, s \rangle \Rightarrow^k s' \text{ implies } \langle S, s \rangle \rightarrow s'. \]

**Proof:** The proof proceeds by induction on the length of the derivation sequence \( \langle S, s \rangle \Rightarrow^k s' \), that is by induction on \( k \).

If \( k = 0 \) then the result holds vacuously.

To prove the induction step we assume that the lemma holds for \( k \leq k_0 \) and we shall then prove that it holds for \( k_0 + 1 \). We proceed by cases on how the first step of \( \langle S, s \rangle \Rightarrow^{k_0+1} s' \) is obtained, that is by inspecting the derivation tree for the first step of computation in the structural operational semantics.

**The case** \([\text{ass}_{sos}]\): Straightforward (and \( k_0 = 0 \)).

**The case** \([\text{skip}_{sos}]\): Straightforward (and \( k_0 = 0 \)).

Clearly, the cases for compound statement forms somehow have to take apart the \( k \) transitions to account for the transitions needed for the constituents.
Taking apart statement composition

**Lemma [2.19]** If \((S_1; S_2, s) \Rightarrow^k s''\) then there exists \(s', k_1, k_2\) such that
\[
(S_1, s) \Rightarrow^{k_1} s',
(S_2, s') \Rightarrow^{k_2} s'' \quad \text{and}
\]
\(k = k_1 + k_2\)

**Proof** We proceed by induction on the number \(k\).

Proof by induction on the length of derivation sequences
Lemma 2.19 If \( \langle S_1; S_2, s \rangle \Rightarrow^k s'' \) then there exists a state \( s' \) and natural numbers \( k_1 \) and \( k_2 \) such that \( \langle S_1, s \rangle \Rightarrow^{k_1} s' \) and \( \langle S_2, s' \rangle \Rightarrow^{k_2} s'' \) where \( k = k_1 + k_2 \).

Proof: The proof is by induction on the number \( k \), that is by induction on the length of the derivation sequence \( \langle S_1; S_2, s \rangle \Rightarrow^k s'' \).

If \( k = 0 \) then the result holds vacuously.

For the induction step we assume that the lemma holds for \( k \leq k_0 \) and we shall prove it for \( k_0 + 1 \). So assume that

\[
\langle S_1; S_2, s \rangle \Rightarrow^{k_0+1} s''
\]

This means that the derivation sequence can be written as

\[
\langle S_1; S_2, s \rangle \Rightarrow \gamma \Rightarrow^{k_0} s''
\]

for some configuration \( \gamma \). Now one of two cases applies depending on which of the two rules \([\text{comp}_{sos}^1]\) and \([\text{comp}_{sos}^2]\) was used to obtain \( \langle S_1; S_2, s \rangle \Rightarrow \gamma \).
In the first case where [\text{comp}^{1}_{sos}] is used we have

\[ \langle S_1; S_2, s \rangle \Rightarrow \langle S'_1; S_2, s' \rangle \]

because

\[ \langle S_1, s \rangle \Rightarrow \langle S'_1, s' \rangle \]

We therefore have

\[ \langle S'_1; S_2, s' \rangle \Rightarrow^{k_0} s'' \]

and the induction hypothesis can be applied to this derivation sequence because it is shorter than the one we started with. This means that there is a state \( s_0 \) and natural numbers \( k_1 \) and \( k_2 \) such that

\[ \langle S'_1, s' \rangle \Rightarrow^{k_1} s_0 \text{ and } \langle S_2, s_0 \rangle \Rightarrow^{k_2} s'' \]

where \( k_1 + k_2 = k_0 \). Using that \( \langle S_1, s \rangle \Rightarrow \langle S'_1, s' \rangle \) and \( \langle S'_1, s' \rangle \Rightarrow^{k_1} s_0 \) we get

\[ \langle S_1, s \rangle \Rightarrow^{k_1+1} s_0 \]

We have already seen that \( \langle S_2, s_0 \rangle \Rightarrow^{k_2} s'' \) and since \( (k_1+1)+k_2 = k_0+1 \) we have proved the required result.
The second possibility is that \([\text{comp}^2_{\text{sos}}]\) has been used to obtain the derivation 
\[\langle S_1; S_2, s \rangle \Rightarrow \gamma.\] Then we have 
\[\langle S_1, s \rangle \Rightarrow s'\]
and \(\gamma\) is \(\langle S_2, s' \rangle\) so that 
\[\langle S_2, s' \rangle \Rightarrow^{k_0} s''\]

The result now follows by choosing \(k_1=1\) and \(k_2=k_0\). \qedhere
\[ \langle S, s \rangle \Rightarrow^k s' \text{ implies } \langle S, s \rangle \rightarrow s'. \text{ cont'd} \]

**The cases** [comp\(^1\)\(_{\text{sos}}\)] and [comp\(^2\)\(_{\text{sos}}\): In both cases we assume that

\[ \langle S_1; S_2, s \rangle \Rightarrow^{k_0+1} s'' \]

We can now apply (Lemma 2.19) and get that there exists a state \( s' \) and natural numbers \( k_1 \) and \( k_2 \) such that

\[ \langle S_1, s \rangle \Rightarrow^{k_1} s' \text{ and } \langle S_2, s' \rangle \Rightarrow^{k_2} s'' \]

where \( k_1 + k_2 = k_0 + 1 \). The induction hypothesis can now be applied to each of these derivation sequences because \( k_1 \leq k_0 \) and \( k_2 \leq k_0 \). So we get

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

Using [comp\(_{\text{ns}}\)] we now get the required \( \langle S_1; S_2, s \rangle \rightarrow s'' \).

Further composites omitted.
End Of Slide Show