FLOLAC 2010

Intro. to Operational Semantics

Kung Chen
National Chengchi University, Taiwan

2010 Formosan Summer School of Logic,
Language and Computation

FLOLAC 2010

Agenda

- Overview: What and Why of Formal Semantics
- Operational Semantics for While
  - Natural Semantics (Big-step)
  - Structured Operational Semantics (SOS, Small-step)
- Extensions of While
  - Abortion
  - Non-determinism and Parallelism
  - Procedures and Blocks

Describing Programs

FLOLAC 2010

- **Syntax**: what sequences of characters constitute programs? Grammars, lexers, parsers, automata theory.

- **Semantics**: what does a program mean (do)? When are two programs *equivalent*? When does a program satisfy its specification?

Semantics: What does a program do?

FLOLAC 2010

- Hard to get it right!

- What does the following C statement print if "x==1"?

```
printf("%d %d\n", x++, ++x);
```

- Can we replace `"f(x)+f(x)"` with `"2*f(x)"`?

What does a program mean?

FLOLAC 2010

- Compile and run
  - Implementation dependencies
  - Not useful for reasoning
- Informal Semantics (Reference manual)
  - Natural language description of PL
- Formal Semantics
  - Description in terms of notation with formally understood meaning

Why Not Informal Semantics?

FLOLAC 2010

- An extract from the Algol 60 report:

"Finally the procedure body, modified as above, is inserted in place of the procedure statement and executed. *If a procedure is called from a place outside the scope of any nonlocal quantity of the procedure body, the conflicts between the identifiers inserted through this process of body replacement and the identifiers whose declarations are valid at the place of the procedure statement or function designator will be avoided through suitable systematic changes of the latter identifiers*."

Why Formal Semantics?

FLOLAC 2010

- precise specification of software (and hardware)
- facilitate reasoning about systems; testing may reveal errors but not their absence
- help in understanding subtle details and ambiguities in apparently clear defining documents (otherwise discovered late – e.g. by implementors; bad situation when an implementation must be treated as language definition)
- subject to mathematical methods e.g. proving programs correct
- form the basis for prototype implementations, e.g. interpreters and compilers; and for software tools

Styles of Formal Semantics

FLOLAC 2010

Denotational: a program's meaning is given abstractly as *an element of some mathematical structure* (some kind of set).

⇒ **Operational**: a program's meaning is given in terms of *the steps of computation* the program makes when you run it.

Axiomatic: a program's meaning is given *indirectly in terms of the collection of properties it satisfies*; these properties are defined via a collection of axioms and rules.

Operational Semantics

FLOLAC 2010

$y := 1;$
$\texttt{while}\ \neg(x = 1)\ \texttt{do}\ (y := x * y; x := x - 1)$

First we assign 1 to y, then we test whether x is 1 or not. If it is then we stop and otherwise we update y to be the product of x and the previous value of y and then we decrement x by one. Now we test whether the new value of x is 1 or not $\cdots$

Two kinds of operational semantics:

- Natural Semantics
- Structural Operational Semantics

Denotational Semantics

FLOLAC 2010

$y := 1;$
$\texttt{while}\ \neg(x = 1)\ \texttt{do}\ (y := x * y; x := x - 1)$

The program computes a partial function from states to states: the final state will be equal to the initial state except that the value of x will be 1 and the value of y will be equal to the factorial of the value of x in the initial state.

Two kinds of denotational semantics:

- Direct Style Semantics
- Continuation Style Semantics

Axiomatic Semantics

FLOLAC 2010

$y := 1;$
$\texttt{while}\ \neg(x = 1)\ \texttt{do}\ (y := x * y; x := x - 1)$

If $x = n$ holds before the program is executed then $y = n!$ will hold when the execution terminates (if it terminates)

Two kinds of axiomatic semantics:

- Partial Correctness
- Total Correctness

Which approach?

FLOLAC 2010

Programming Language

$\downarrow$

Semantics

- natural semantics
- structural operational semantics
- direct style denotational semantics
- continuation style denotational semantics
- partial correctness axiomatic semantics
- total correctness axiomatic semantics

Selection criteria

FLOLAC 2010

- constructs of the language
  - imperative
  - functional
  - concurrent/parallel
  - object oriented
  - non-deterministic
  - ...

- what is the semantics used for
  - understanding the language
  - verification of programs
  - prototyping
  - compiler construction
  - program analysis
  - ...

This Course Unit

FLOLAC 2010

- Is based on the first three chapters of the book:
  - Semantics with Applications: an Appetizer, by Nielson & Nielson
- Uses a simple imperative language: **While** to introduce **operational semantics**
- There will be some mathematics along the way:
  - mathematical induction; and
  - Structural induction.

Example Language: While

2010

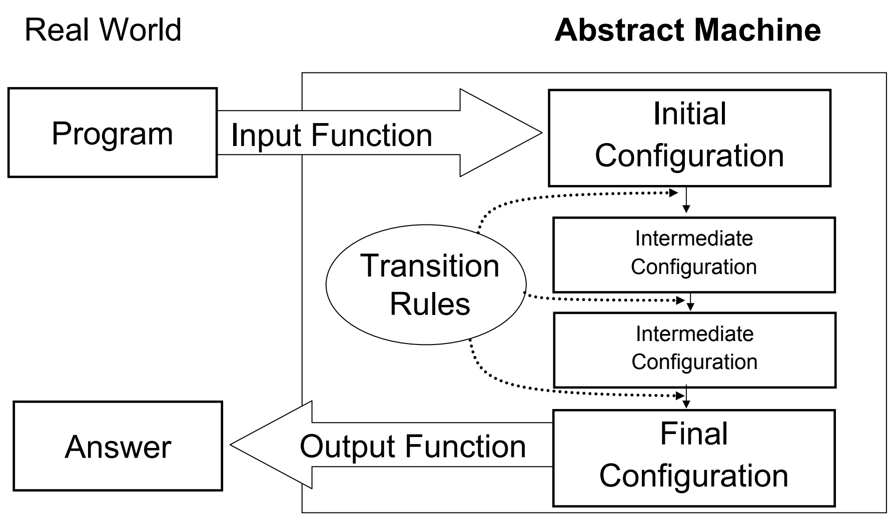
•A simple imperative language without procedures.

$$
\begin{aligned}
S \ ::= \ & x := a \mid \texttt{skip} \mid S_1 ; S_2 \\
\mid \ & \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \\
\mid \ & \texttt{while}\ b\ \texttt{do}\ S \\
\mid \ & \texttt{repeat}\ S\ \texttt{until}\ b
\end{aligned}
$$

•And some extensions of **While**

Operational Semantics

2010



Source: D. Evans

Operational Semantics

FLOLAC 2010

Operational semantics works with configurations of the form

$$\langle \mathit{control}, \mathit{data} \rangle.$$

program

Roughly:

control – "where are we", *data* – the values of program variables.

control may be absent (final configuration).

Structural Operational Semantics

Sequences of configurations, $\mathit{conf}_1 \Rightarrow \mathit{conf}_2 \Rightarrow \cdots$.

(Small step semantics.)

Natural Semantics (big step semantics)

$\langle \mathit{control}, \mathit{data} \rangle \rightarrow \mathit{data}'$ in one step.

Approach

FLOLAC 2010

concrete syntax	LL(1)/LALR(1) grammar concrete syntax trees
↓	
abstract syntax	syntactic categories abstract syntax trees
↓	
semantics	semantic categories semantic definitions

Acknowledgement

FLOLAC 2010

- Most of the following slides are taken from
  - Slides by Prof. Hannie Riis Nielson: Introduction to Semantics
  - Slides by Prof. Ralf Lammel: Programming Paradigms and Formal Semantics

FLOLAC 2010

Syntax of While

Syntactic Categories

FLOLAC 2010

- Numerals — $n \in$ Num
- Variables — $x \in$ Var
- Arithmetic expressions — $a \in$ AExp

not further specified

$$a \quad ::= \quad n \mid x \mid a_1 + a_2 \mid a_1 \star a_2 \mid a_1 - a_2$$

- Boolean expressions — $b \in$ BExp

$$b \quad ::= \quad \texttt{true} \mid \texttt{false} \mid a_1 = a_2 \mid a_1 \leq a_2 \mid \neg b \mid b_1 \wedge b_2$$

- Statements — $S \in$ Stm

$$S \quad ::= \quad x := a \mid \texttt{skip} \mid S_1; S_2 \mid \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \mid \texttt{while}\ b\ \texttt{do}\ S \mid \texttt{repeat}\ S\ \texttt{until}\ b$$

Hanne Riis Nielson

OP semantics

21

Abstract vs. Concrete Syntax

FLOLAC 2010

- Abstract Syntax

  focusses on the *structure* of expressions, statements, etc and ignores the scanning and parsing aspects of the syntax

$$a ::= n \mid x \mid a_1 + a_2 \mid a_1 * a_2$$

- Concrete Syntax

  deals with scanning and parsing aspects

A ::= T+ A | T
T ::= F * T | F
F ::= N | X | (A)

N: digit+
X: letter (digit | letter)*

Hanne Riis Nielson

OP semantics

22

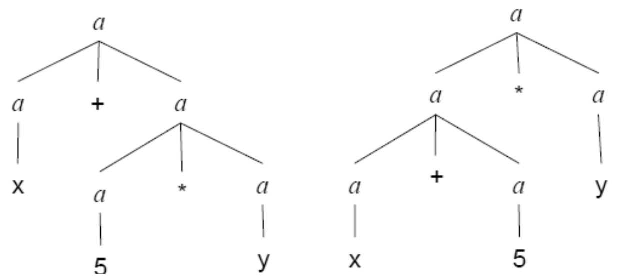
Example: x+(5*y)

FLOLAC 2010

$$a ::= n \mid x \mid a_1 + a_2 \mid a_1 * a_2$$

- Abstract syntax:
  - formalises the allowable parse trees
  - we use parentheses to disambiguate the syntax
  - we introduce defaults as e.g. * binds closer than +

default
x + (5 * y)



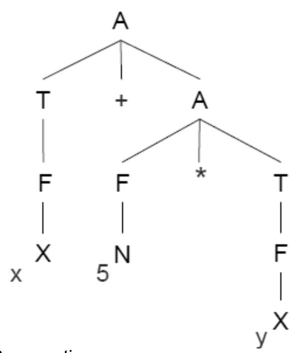
write
(x + 5) * y

Example: x+(5*y)

FLOLAC 2010

- Concrete syntax
  - parantheses disambiguate the syntax
  - the grammar captures aspects like the precedence and associativity rules

A ::= T+ A | T
T ::= F * T | F
F ::= N | X | (A)



Other Ambiguities

FLOLAC 2010

$$S ::= x := a \mid \texttt{skip} \mid S_1 \,;\, S_2 \mid \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \mid \texttt{while}\ b\ \texttt{do}\ S$$

x := 1; y := 2; z := 3

x := 1; (y := 2; z := 3)

(x := 1; y := 2); z := 3

if x < y then x := 1; y := 2 else x := 3; y := 4

if x < y then (x := 1; y := 2) else x := 3; y := 4

if x < y then x := 1; y := 2 else (x := 3; y := 4)

while x < y do x := x+1; y := 0

while x < y do (x := x+1; y := 0)

Example Programs

FLOLAC 2010

- factorial program:
  - if x = *n* initially then y= *n!* when the program terminates
  - y := 1; while ¬(x=1) do (y:=y*x; x:=x-1)
- power function:

  Exercise 1.2
  - if x = *n* and y = *m* initially then z = n^m when the program terminates
  - write the program in the while language

FLOLAC 2010

Semantics of Expressions

Note: Not really an OP approach;
Semantics of Statements will be formulated in an OP way.

FLOLAC 2010

Memory Model: State

- the value of x+5*y depends on the values of the variables x and y
- these are determined by the current state
- operations on states:

lookup in a state: $s\ x$

update a state: $s' = s[y \mapsto n]$

$$s'\ x = \begin{cases} s\ x & \text{if } x \neq y \\ n & \text{if } x = y \end{cases}$$

variables

semantic values: numbers

x	2
y	4
z	0

the value of x+5*y is 22:

$\mathcal{A}$ [x+5*y]s = s(x)+5*s(y)
= 2+5*4
= 22

Semantic Functions

FLOLAC 2010

- $\mathcal{A}$: AExp $\rightarrow$ (State $\rightarrow$ Z)

  for each arithmetic expression a and each state s the function determines the value (a number) $\mathcal{A}[a]s$ of a

- $\mathcal{B}$: BExp $\rightarrow$ (State $\rightarrow$ T)

  for each boolean expression b and each state s the function determines the value (true or false) $\mathcal{B}[b]s$ of b

Hanne Riis Nielson

OP semantics

29

$\mathcal{A}$: AExp $\rightarrow$ (State $\rightarrow$ Z)

FLOLAC 2010

one clause for each of the different forms of arithmetic expressions

$\mathcal{N}$: Num $\rightarrow$ Z
from numerals (syntax) to numbers (semantics)

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

symbols of syntax

semantic operators

Hanne Riis Nielson

OP semantics

30

$\mathcal{B}$: BExp → (State → T)

FLOLAC 2010

$$
\begin{aligned}
\mathcal{B}[\![\texttt{true}]\!]s &= \mathbf{tt} \\
\mathcal{B}[\![\texttt{false}]\!]s &= \mathbf{ff} \\
\mathcal{B}[\![a_1 = a_2]\!]s &= \begin{cases} \mathbf{tt} & \text{if } \mathcal{A}[\![a_1]\!]s = \mathcal{A}[\![a_2]\!]s \\ \mathbf{ff} & \text{if } \mathcal{A}[\![a_1]\!]s \neq \mathcal{A}[\![a_2]\!]s \end{cases} \\
\mathcal{B}[\![a_1 \leq a_2]\!]s &= \begin{cases} \mathbf{tt} & \text{if } \mathcal{A}[\![a_1]\!]s \leq \mathcal{A}[\![a_2]\!]s \\ \mathbf{ff} & \text{if } \mathcal{A}[\![a_1]\!]s > \mathcal{A}[\![a_2]\!]s \end{cases} \\
\mathcal{B}[\![\neg\, b]\!]s &= \begin{cases} \mathbf{tt} & \text{if } \mathcal{B}[\![b]\!]s = \mathbf{ff} \\ \mathbf{ff} & \text{if } \mathcal{B}[\![b]\!]s = \mathbf{tt} \end{cases} \\
\mathcal{B}[\![b_1 \wedge b_2]\!]s &= \begin{cases} \mathbf{tt} & \text{if } \mathcal{B}[\![b_1]\!]s = \mathbf{tt} \text{ and } \mathcal{B}[\![b_2]\!]s = \mathbf{tt} \\ \mathbf{ff} & \text{if } \mathcal{B}[\![b_1]\!]s = \mathbf{ff} \text{ or } \mathcal{B}[\![b_2]\!]s = \mathbf{ff} \end{cases}
\end{aligned}
$$

(truth values)

one clause for each of the different forms of boolean expressions

The rules of the game

FLOLAC 2010

- the syntactic category is specified by giving
  - the basic elements
  - the composite elements; these have a unique decomposition into their immediate constituents

$$
\begin{aligned}
a &::= \boxed{n \mid x} \mid a_1 + a_2 \mid a_1 \star a_2 \mid a_1 - a_2 \\
b &::= \boxed{\texttt{true} \mid \texttt{false} \mid a_1 = a_2 \mid a_1 \leq a_2} \mid \neg b \mid b_1 \wedge b_2
\end{aligned}
$$

The rules of the game

FLOLAC 2010

- The semantics is then defined by a compositional definition of a function:
  - there is a semantic clause for each of the basic elements of the syntactic category
  - there is a semantic clause for each of the ways for constructing composite elements; the clause is defined in terms of the semantics for the immediate constituents of the elements

$$\left.\begin{array}{lcl}\mathcal{A}[\![n]\!]s & = & \mathcal{N}[\![n]\!] \\ \mathcal{A}[\![x]\!]s & = & s\ x\end{array}\right\}\ \text{basic elements}$$

$$\left.\begin{array}{lcl}\mathcal{A}[\![a_1 + a_2]\!]s & = & \mathcal{A}[\![a_1]\!]s + \mathcal{A}[\![a_2]\!]s \\ \mathcal{A}[\![a_1 \star a_2]\!]s & = & \mathcal{A}[\![a_1]\!]s \star \mathcal{A}[\![a_2]\!]s \\ \mathcal{A}[\![a_1 - a_2]\!]s & = & \mathcal{A}[\![a_1]\!]s - \mathcal{A}[\![a_2]\!]s\end{array}\right\}\ \text{composite elements}$$

A simple result

FLOLAC 2010

- We want to formalise the fact that the value of an arithmetic expression only depends on the values of the variables occurring in it
- **Definition:** FV(*a*) is the set of free variables in the arithmetic expression *a*

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

A simple result and its proof

FLOLAC 2010

- **Lemma**: Assume that *s* and *s'* are states satisfying *s*(*x*) = *s'*(*x*) for all *x* in FV(*a*). Then $\mathcal{A}[a]s = \mathcal{A}[a]s'$.

- **Proof**: by Structural Induction
  - case *n*
  - case *x*
  - case $a_1 + a_2$
  - case $a_1 * a_2$
  - case $a_1 - a_2$

Structural Induction

FLOLAC 2010

To prove a property of all the elements of the syntactic category do the following:

- Prove that the property holds for all the basis elements of the syntactic category.
- 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.

A substitution result

FLOLAC 2010

- We want to formalise the fact that a substitution within an expression can be mimicked by a similar change of the state.
- **Definition**: Replacing all occurrences of *y* within *a* with a_0:

$$
\begin{aligned}
n[y\mapsto a_0] &= n\\
x[y\mapsto a_0] &= \begin{cases} a_0 & \text{if } x = y\\ x & \text{if } x \neq y\end{cases}\\
(a_1 + a_2)[y\mapsto a_0] &= (a_1[y\mapsto a_0]) + (a_2[y\mapsto a_0])\\
(a_1 \star a_2)[y\mapsto a_0] &= (a_1[y\mapsto a_0]) \star (a_2[y\mapsto a_0])\\
(a_1 - a_2)[y\mapsto a_0] &= (a_1[y\mapsto a_0]) - (a_2[y\mapsto a_0])
\end{aligned}
$$

A substitution result and its proof

FLOLAC 2010

- **Lemma**: Let

$$(s[y\mapsto v])\ x = \begin{cases} v & \text{if } x = y\\ s\ x & \text{if } x \neq y\end{cases}$$

- then for all states *s*

$$\mathcal{A}[\![a[y\mapsto a_0]]\!]s = \mathcal{A}[\![a]\!](s[y\mapsto \mathcal{A}[\![a_0]\!]s])$$

- **Proof**: Exercise 1.13

FLOLAC 2010

Semantics of Statements

Updating the states

FLOLAC 2010

- An assignment updates the state

state before executing z := x+5*y

x	2
y	4
z	0

- general formulation:

$$\langle x := a,\ s\rangle \rightarrow \underbrace{s[x \mapsto \mathcal{A}[\![a]\!]s]}$$

state before executing $x := a$

state after executing $x := a$

state after executing z := x+5*y

x	2
y	4
z	22

Two kinds of semantics

FLOLAC 2010

- Natural semantics (NS)
  - given a statement and a state in which it has to be executed, what is the resulting state (if it exists)? **(Big-step)**
- Structural operational semantics (SOS)
  - given a statement and a state in which it has to be executed, what is the next step of the computation (if it exists)? **(Small-step)**

Natural semantics

FLOLAC 2010

- the result of executing the assignment *x* := *a* in the state *s* is the state *s* updated such that *x* gets the value of *a*
- the result of executing the skip statement in the state *s* is simply the state *s*

$$\langle x := a,\ s\rangle \to s[x \mapsto \mathcal{A}[\![a]\!]s]$$

$$\langle \texttt{skip},\ s\rangle \to s$$

Axiom schemas:
they can be instantiated for particular choices of x, a and s

Natural semantics

FLOLAC 2010

- the result of executing S_1; S_2 from the state s is obtained by first executing the S_1 from s to obtain its resulting state s' and then to execute S_2 from that state to obtain its resulting state s'' and that will be the resulting state for S_1; S_2

$$\frac{\langle S_1,\ s\rangle \to s',\ \langle S_2,\ s'\rangle \to s''}{\langle S_1;S_2,\ s\rangle \to s''}$$

a rule with
- two premises
and
- one conclusion

Building a derivation

FLOLAC 2010

axiom schemas

$\langle \texttt{skip},\ s\rangle \to s$ $\qquad$ $\langle x := a,\ s\rangle \to s[x \mapsto \mathcal{A}[\![a]\!]s]$

assume x is 0 in s_0

instances

$\langle \texttt{skip},\ s_0\rangle \to s_0$ $\qquad$ $\langle \texttt{x := x+1},\ s_0\rangle \to s_0[\texttt{x}\mapsto\mathbf{1}]$

the state that is as s_0 except that x is 1

$$\frac{\langle \texttt{skip},\ s_0\rangle \to s_0,\ \langle \texttt{x := x+1},\ s_0\rangle \to s_0[\texttt{x}\mapsto\mathbf{1}]}{\langle \texttt{skip; x := x+1},\ s_0\rangle \to s_0[\texttt{x}\mapsto\mathbf{1}]}$$

instance of rule: the premises are satisfied

$$\frac{\langle S_1,\ s\rangle \to s',\ \langle S_2,\ s'\rangle \to s''}{\langle S_1;S_2,\ s\rangle \to s''}$$

Natural semantics

FLOLAC 2010

- The result of executing if b then S_1 else S_2 from state s depends on the value of b in state s:
  - If it is **tt** then the result is the resulting state of S_1
  - If it is **ff** then the result is the resulting state of S_2

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

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

side conditions

must be computable

Natural semantics

FLOLAC 2010

- The result of executing while b do S from state s depends on the value of b in state s:
  - If it is **tt** then we first execute S from s to obtain its resulting state s' and then repeat the execution of while b do S but from s' in order to obtain its resulting state s'' which then will be the overall resulting state
  - If it is **ff** then the resulting state is simply s

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

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

side conditions

Summary: natural semantics

FLOLAC 2010

$$\langle x := a\,,\ s\rangle \to s[x \mapsto \mathcal{A}[\![a]\!]s]$$

$$\langle \texttt{skip}\,,\ s\rangle \to s$$

$$\frac{\langle S_1\,,\ s\rangle \to s' \quad \langle S_2\,,\ s'\rangle \to s''}{\langle S_1; S_2\,,\ s\rangle \to s''}$$

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

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

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

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

Example

FLOLAC 2010

$s_{ij}(y) = i$, $s_{ij}(x) = j$
$s = s_{03}$

Factotial, x==3

$$\langle \texttt{y:=y*x},\ s_{32}\rangle \to s_{62} \qquad \langle \texttt{x:=x-1},\ s_{62}\rangle \to s_{61}$$

$$\langle \texttt{y:=y*x; x:=x-1},\ s_{32}\rangle \to s_{61} \qquad \langle \texttt{while}\ \neg(\texttt{x=1})\ \texttt{do}\ (\texttt{y:=y*x; x:=x-1}),\ s_{61}\rangle \to s_{61}$$

$$\langle \texttt{y:=y*x},\ s_{13}\rangle \to s_{33} \qquad \langle \texttt{x:=x-1},\ s_{33}\rangle \to s_{32}$$

$$\langle \texttt{y:=y*x; x:=x-1},\ s_{13}\rangle \to s_{32} \qquad \langle \texttt{while}\ \neg(\texttt{x=1})\ \texttt{do}\ (\texttt{y:=y*x; x:=x-1}),\ s_{32}\rangle \to s_{61}$$

$$\langle \texttt{y:=1},\ s\rangle \to s_{13} \qquad \langle \texttt{while}\ \neg(\texttt{x=1})\ \texttt{do}\ (\texttt{y:=y*x; x:=x-1}),\ s_{13}\rangle \to s_{61}$$

$$\langle \texttt{y:=1; while}\ \neg(\texttt{x=1})\ \texttt{do}\ (\texttt{y:=y*x; x:=x-1}),\ s\rangle \to s_{61}$$

Exercise

FLOLAC 2010

- Consider the statement

  z := 0; while y ≤ x do (z := z+1; x := x-y)

  Construct a derivation tree for the statement when executed in a state where x has the value 17 and y has the value 5.

FLOLAC 2010

Terminology

Derivation trees

FLOLAC 2010

- When we use the axioms and rules to derive a transition $\langle S,s\rangle \to s'$ we obtain a derivation tree:
  - the *root* of the tree is $\langle S,s\rangle \to s'$
  - the leaves of the tree are instances of the axioms
  - the *internal nodes* of the tree are the conclusions of instances of the rules; they have the corresponding instances of their premises as immediate sons
- The execution of S from s
  - *terminates* if there is a state s' such that $\langle S,s\rangle \to s'$
  - *loops* if there is *no* state s' such that $\langle S,s\rangle \to s'$

Exercise

FLOLAC 2010

- Consider the following statements
  - while $\neg$ (x = 1) do (y := y*x; x := x-1)
  - while 1 ≤ x do (y := y*x; x := x-1)
  - while true do skip

  For each statement determine whether or not it always terminates or it always loops. Argue for your answer using the axioms and rules of the NS.

Semantic equivalence

FLOLAC 2010

- **Definition**: Two statements S_1 and S_2 are semantically equivalent if for all states *s* and *s'*

  $\langle S_1, s\rangle \rightarrow s'$ if and only if $\langle S_2, s\rangle \rightarrow s'$

- **Lemma**: while *b* do *S* and if *b* then (*S*; while *b* do *S*) else skip are semantically equivalent

Property of the Semantics

FLOLAC 2010

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: (**) ⇒ (*)

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

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

 27

Lemma 2.5: ***while b do S*** and **if b then (S; while b do S) else skip** are semantically equivalent

FLOLAC 2010

Proof part 1:

Assume
$\mathcal{B}$ [b]s = tt

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

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

Lemma 2.5: **while b do S** and **if b then (S; while b do S) else skip** are semantically equivalent

FLOLAC 2010

Assume
$\mathcal{B}$ [b]s = ff

$$\langle \texttt{while}\ b\ \texttt{do}\ S,\ s\rangle \to s$$

s=s'' must
be the case

Proof part 2:

s=s'' must
be the case

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

Exercise

- Prove that $(S_1; S_2); S_3$ and $S_1; (S_2; S_3)$ are semantically equivalent

Proof principles for natural semantics

Deterministic semantics

FLOLAC 2010

- **Definition**: the natural semantics is deterministic if for all statements *S* and states *s*, *s'* and *s''*

$$\langle S,\, s\rangle \to s' \text{ and } \langle S,\, s\rangle \to s'' \text{ imply } s' = s''$$

- **Lemma**: the natural semantics of the while language is deterministic

- **Proof**: by induction on the shape of the derivation tree

Induction on the shape of derivation trees

FLOLAC 2010

To prove a property of all the derivation trees of a natural semantics do the following:

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

Why not induction on the structure of programs?

FLOLAC 2010

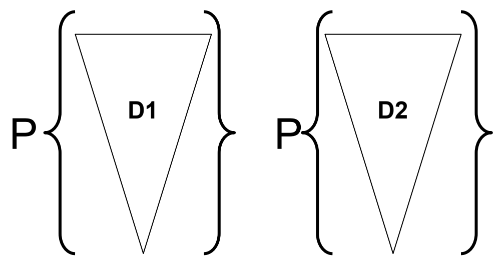
- Because the semantics for While-statement is self-referential.

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

Induction of the shape of derivation trees: P(.) holds

FLOLAC 2010

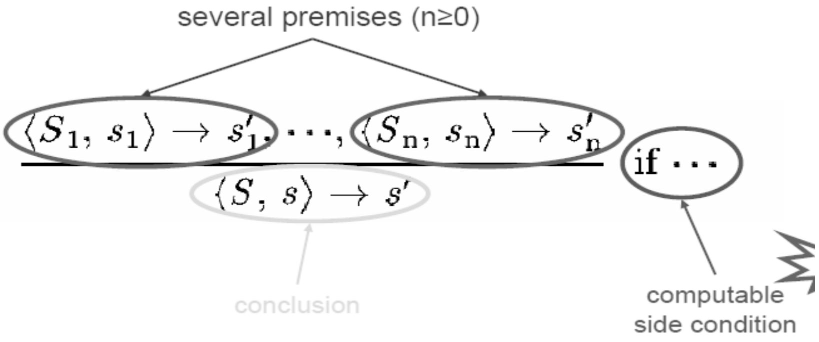
(I) Axioms: induction base
(II)Inference rules



$$P\,(\,\langle S,\ s\rangle \rightarrow s'\,)$$

Induction of the shape of derivation trees: P(.) holds

FLOLAC 2010



- If we have derivation trees that matches the premises and if the side condition is fulfilled
- then we can construct a derivation tree for the conclusion

Proof of Determinacy

FLOLAC 2010

The case [comp$_{ns}$]: 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$_{ns}$] 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.

Summary: Natural semantics

FLOLAC 2010

simple derivation trees:

$$\langle x := a\ ,\ s\rangle \rightarrow s[x \mapsto \mathcal{A}[\![a]\!]s]$$

$$\langle \mathtt{skip}\ ,\ s\rangle \rightarrow s$$

composite derivation trees:

$$\frac{\langle S_1\ ,\ s\rangle \rightarrow s' \quad \langle S_2\ ,\ s'\rangle \rightarrow s''}{\langle S_1; S_2\ ,\ s\rangle \rightarrow s''}$$

$$\frac{\langle S_1\ ,\ s\rangle \rightarrow s'}{\langle \mathtt{if}\ b\ \mathtt{then}\ S_1\ \mathtt{else}\ S_2\ ,\ s\rangle \rightarrow s'} \quad \text{if } \mathcal{B}[\![b]\!]s = \mathbf{tt}$$

$$\frac{\langle S_2\ ,\ s\rangle \rightarrow s'}{\langle \mathtt{if}\ b\ \mathtt{then}\ S_1\ \mathtt{else}\ S_2\ ,\ s\rangle \rightarrow s'} \quad \text{if } \mathcal{B}[\![b]\!]s = \mathbf{ff}$$

$$\frac{\langle S\ ,\ s\rangle \rightarrow s' \quad \langle \mathtt{while}\ b\ \mathtt{do}\ S\ ,\ s'\rangle \rightarrow s''}{\langle \mathtt{while}\ b\ \mathtt{do}\ S\ ,\ s\rangle \rightarrow s''} \quad \text{if } \mathcal{B}[\![b]\!]s = \mathbf{tt}$$

$$\langle \mathtt{while}\ b\ \mathtt{do}\ S\ ,\ s\rangle \rightarrow s \quad \text{if } \mathcal{B}[\![b]\!]s = \mathbf{ff}$$

Summary

FLOLAC 2010

- **Summary**: *Big-step operational semantics*
  - *Models relations between syntax, states, values.*
  - *Uses deduction rules (conclusion, premises).*
  - *Computations are derivation trees.*
- **Prepping**: *"Semantics with applications"*
  - *Chapter 1 and Chapter 2.1*
- **Outlook**:
  - *Small-step semantics*
  - *More properties and proofs*
  - *Language extensions of* **While**

Homework

FLOLAC 2010

The exercise session today:
Natural Semantics
for
the repeat construct

$$S ::= x := a \mid \texttt{skip} \mid S_1; S_2 \mid \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2$$
$$\mid \cancel{\texttt{while } b \texttt{ do } S} \mid \texttt{repeat } S \texttt{ until } b$$

Homework

FLOLAC 2010

- Specify the semantics of the construct
  repeat *S* until *b*

  The specification is not allowed to rely on the existence of the while construct in the language.

  Prove that repeat *S* until *b* is semantically equivalent to *S*; if *b* then skip else (repeat *S* until *b*) (Optional)

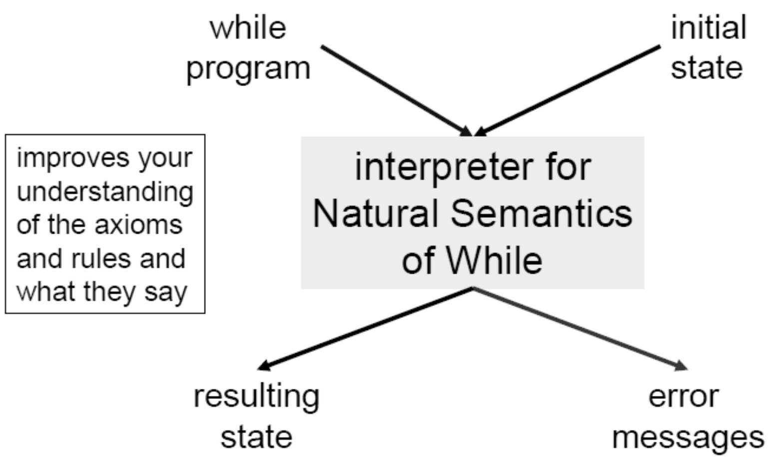
FLOLAC 2010

Programming Exercise

Code Skelton in ML will be provided

Goal

FLOLAC 2010



Hanne Riis Nielson

Syntax of While in ML

FLOLAC 2010

each syntactic category gives rise to a data type

```
type NUM = string
type VAR = string

datatype AEXP = Num   of NUM
              | Var   of VAR
              | Add   of AEXP * AEXP
              | Mult  of AEXP * AEXP
              | Sub   of AEXP * AEXP

datatype BEXP = tt
              | ff
              | …

datatype STM  = Ass   of VAR  * AEXP
              | Skip
              | …
```

Example: y := y * x becomes `Ass ("y", Mult (Var "y", Var "x"))`

OP semantics

71

Expressions

FLOLAC 2010

Expressions

State = Var → Z

$\mathcal{N}$: Num → Z

$\mathcal{A}$: AExp → (State → Z)

$\mathcal{B}$: BExp → (State → T)

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

each semantic function gives rise to a SML function

```
type STATE = VAR -> int

(* N : NUM -> int *)

fun N n = valOf (Int.fromString n)

(* A: AEXP -> STATE -> int *)

fun A (Num n) s         = N n
  | A (Var x) s         = s x
  | A (Add (a1,a2)) s   = A a1 s + A a2 s
  | A ...
```

Hanne Riis Nielson

72

Natural Semantics

2010

$\langle x := a\,,\, s\rangle \to s[x \mapsto \mathcal{A}[\![a]\!]s]$

$\langle \texttt{skip}\,,\, s\rangle \to s$

$$\frac{\langle S_1\,,\, s\rangle \to s' \quad \langle S_2\,,\, s'\rangle \to s''}{\langle S_1; S_2\,,\, s\rangle \to s''}$$

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

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

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

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

the transition relation gives rise to a function in SML – why does that work, by the way?

```
datatype CONFIG
   = Inter of STM * STATE
   | Final of STATE
```

```
fun update x a s = ...

fun NS (Inter ((Ass (x,a)), s) )      = Final ...
 |  NS (Inter (Skip, s))              = Final s
 |  NS ..
```


Exercise

2010

- Complete the SML implementation
- Test the implementations on programs like
  - y := 1; while ¬(x = 1) do (y := y * x; x := x – 1)
  - z := 0; while y ≤ x do (z := z+1; x := x-y)
  - while ¬ (x = 1) do (y := y*x; x := x-1)
  - while 1 ≤ x do (y := y*x; x := x-1)
  - while true do skip

  using a number of different states
- Extend the implementation to include the repeat construct

x = 1

let x = 1 in ...

x(1).

!x(1)

x.set(1)

Programming Paradigms and Formal Semantics

Small-Step Operational Semantics

Ralf Lämmel

Structured operational semantics: describe how the individual steps of the computation take place.

Transition system: $(\Gamma, T, \Rightarrow)$

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

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:
  $$(S, s) \Rightarrow s'$$

SOS (statements)

$[\text{ass}_{\text{sos}}]$	$\langle x := a,\ s\rangle \Rightarrow s[x \mapsto \mathcal{A}[\![a]\!]s]$
$[\text{skip}_{\text{sos}}]$	$\langle \texttt{skip},\ s\rangle \Rightarrow s$
$[\text{comp}^{1}_{\text{sos}}]$	$\dfrac{\langle S_1,\ s\rangle \Rightarrow \langle S_1',\ s'\rangle}{\langle S_1;S_2,\ s\rangle \Rightarrow \langle S_1';S_2,\ s'\rangle}$
$[\text{comp}^{2}_{\text{sos}}]$	$\dfrac{\langle S_1,\ s\rangle \Rightarrow s'}{\langle S_1;S_2,\ s\rangle \Rightarrow \langle S_2,\ s'\rangle}$
$[\text{if}^{\text{tt}}_{\text{sos}}]$	$\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2,\ s\rangle \Rightarrow \langle S_1,\ s\rangle$ if $\mathcal{B}[\![b]\!]s = \mathbf{tt}$
$[\text{if}^{\text{ff}}_{\text{sos}}]$	$\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2,\ s\rangle \Rightarrow \langle S_2,\ s\rangle$ if $\mathcal{B}[\![b]\!]s = \mathbf{ff}$
$[\text{while}_{\text{sos}}]$	$\langle \texttt{while}\ b\ \texttt{do}\ S,\ s\rangle \Rightarrow \langle \texttt{if}\ b\ \texttt{then}\ (S;\ \texttt{while}\ b\ \texttt{do}\ S)\ \texttt{else}\ \texttt{skip},\ s\rangle$

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

Illustration of transitions

Derivation sequence (many transitions)

$$
\begin{aligned}
&\langle (\mathrm{z} := \mathrm{x};\ \mathrm{x} := \mathrm{y});\ \mathrm{y} := \mathrm{z},\ s_0\rangle\\
&\qquad \Rightarrow \langle \mathrm{x} := \mathrm{y};\ \mathrm{y} := \mathrm{z},\ s_0[\mathrm{z}\mapsto\mathbf{5}]\rangle\\
&\qquad \Rightarrow \langle \mathrm{y} := \mathrm{z},\ (s_0[\mathrm{z}\mapsto\mathbf{5}])[\mathrm{x}\mapsto\mathbf{7}]\rangle\\
&\qquad \Rightarrow ((s_0[\mathrm{z}\mapsto\mathbf{5}])[\mathrm{x}\mapsto\mathbf{7}])[\mathrm{y}\mapsto\mathbf{5}]
\end{aligned}
$$

Derivation tree (for each single step)

$$
\dfrac{\dfrac{\langle \mathrm{z} := \mathrm{x},\ s_0\rangle \Rightarrow s_0[\mathrm{z}\mapsto\mathbf{5}]}{\langle \mathrm{z} := \mathrm{x};\ \mathrm{x} := \mathrm{y},\ s_0\rangle \Rightarrow \langle \mathrm{x} := \mathrm{y},\ s_0[\mathrm{z}\mapsto\mathbf{5}]\rangle}}{\langle (\mathrm{z} := \mathrm{x};\ \mathrm{x} := \mathrm{y});\ \mathrm{y} := \mathrm{z},\ s_0\rangle \Rightarrow \langle \mathrm{x} := \mathrm{y};\ \mathrm{y} := \mathrm{z},\ s_0[\mathrm{z}\mapsto\mathbf{5}]\rangle}
$$

Statement execution

- Start configuration: statement *S*, state *s*
- Execution ...
  - terminates iff there is a finite derivation sequence starting from ⟨*S*,*s*⟩
  - loops iff there is an infinite derivation sequence starting from ⟨*S*,*s*⟩
  - terminates successfully if ⟨*S*,*s*⟩ ⇒* *s*' for some *s*'.

Big-step style

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

Small-step style

$$[\text{comp}^1_{\text{sos}}] \qquad \frac{\langle S_1,\, s\rangle \Rightarrow \langle S'_1,\, s'\rangle}{\langle S_1;S_2,\, s\rangle \Rightarrow \langle S'_1;S_2,\, s'\rangle}$$

$$[\text{comp}^2_{\text{sos}}] \qquad \frac{\langle S_1,\, s\rangle \Rightarrow s'}{\langle S_1;S_2,\, s\rangle \Rightarrow \langle S_2,\, s'\rangle}$$

Properties of the semantics

Lemma [2.19]

If $(S_1; S_2, s) \Rightarrow^k s''$ then
there exists s', k_1 and k_2 such that
$(S_1, s) \Rightarrow^{k_1} s'$,
$(S_2, s') \Rightarrow^{k_2} s''$ and
$k = k_1 + k_2$

Proof

We proceed by induction on the number k.

Proof by induction on the length of derivation sequences

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

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.

Lemma 2.19 If $\langle S_1;S_2,\ s\rangle \Rightarrow^{\mathrm{k}} s''$ then there exists a state s' and natural numbers k_1 and k_2 such that $\langle S_1,\ s\rangle \Rightarrow^{\mathrm{k}_1} s'$ and $\langle S_2,\ s'\rangle \Rightarrow^{\mathrm{k}_2} s''$ where $\mathrm{k} = \mathrm{k}_1+\mathrm{k}_2$.

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

If $\mathrm{k} = 0$ then the result holds vacuously.

For the induction step we assume that the lemma holds for $\mathrm{k} \leq \mathrm{k}_0$ and we shall prove it for k_0+1. So assume that

$$\langle S_1;S_2,\ s\rangle \Rightarrow^{\mathrm{k}_0+1} s''$$

This means that the derivation sequence can be written as

$$\langle S_1;S_2,\ s\rangle \Rightarrow \gamma \Rightarrow^{\mathrm{k}_0} s''$$

for some configuration γ. Now one of two cases applies depending on which of the two rules $[\mathrm{comp}^1_{\mathrm{sos}}]$ and $[\mathrm{comp}^2_{\mathrm{sos}}]$ was used to obtain $\langle S_1;S_2,\ s\rangle \Rightarrow \gamma$.

$$\frac{\langle S_1,\ s\rangle \Rightarrow \langle S_1',\ s'\rangle}{\langle S_1;S_2,\ s\rangle \Rightarrow \langle S_1';S_2,\ s'\rangle}$$

In the first case where $[\mathrm{comp}^1_{\mathrm{sos}}]$ 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

$$\langle S_1';S_2,\ s'\rangle \Rightarrow^{\mathrm{k}_0} s''$$

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

$$\langle S_1',\ s'\rangle \Rightarrow^{\mathrm{k}_1} s_0 \text{ and } \langle S_2,\ s_0\rangle \Rightarrow^{\mathrm{k}_2} s''$$

where $\mathrm{k}_1+\mathrm{k}_2=\mathrm{k}_0$. Using that $\langle S_1,\ s\rangle \Rightarrow \langle S_1',\ s'\rangle$ and $\langle S_1',\ s'\rangle \Rightarrow^{\mathrm{k}_1} s_0$ we get

$$\langle S_1,\ s\rangle \Rightarrow^{\mathrm{k}_1+1} s_0$$

We have already seen that $\langle S_2,\ s_0\rangle \Rightarrow^{\mathrm{k}_2} s''$ and since $(\mathrm{k}_1+1)+\mathrm{k}_2 = \mathrm{k}_0+1$ we have proved the required result.

$$\frac{\langle S_1,\, s\rangle \Rightarrow s'}{\langle S_1;S_2,\, s\rangle \Rightarrow \langle S_2,\, s'\rangle}$$

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 γ is $\langle S_2,\, s'\rangle$ so that

$$\langle S_2,\, s'\rangle \Rightarrow^{\text{k}_0} s''$$

The result now follows by choosing k_1=1 and k_2=k_0. $\square$

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

Equivalence of semantics

$\mathcal{S}_{\text{ns}}$: **Stm** $\rightarrow$ (**State** $\hookrightarrow$ **State**)

$\mathcal{S}_{\text{sos}}$: **Stm** $\rightarrow$ (**State** $\hookrightarrow$ **State**)

$$\mathcal{S}_{\text{ns}}[\![S]\!]s = \begin{cases} s' & \text{if } \langle S,\, s\rangle \rightarrow s' \\ \underline{\text{undef}} & \text{otherwise} \end{cases}$$

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

Theorem 2.26 For every statement S of **While** we have $\mathcal{S}_{\text{ns}}[\![S]\!] = \mathcal{S}_{\text{sos}}[\![S]\!]$.

Theorem 2.26 For every statement S of **While** we have $\mathcal{S}_{\rm ns}[\![S]\!] = \mathcal{S}_{\rm sos}[\![S]\!]$.

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 $\mathcal{S}_{\rm ns}[\![S]\!] = \mathcal{S}_{\rm sos}[\![S]\!]$.

Lemma 2.27 For every statement S of **While** and states s and s' we have

$$\langle S,\, s\rangle \to s' \text{ implies } \langle S,\, s\rangle \Rightarrow^* s'.$$

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

$$\langle S,\, s\rangle \Rightarrow^{\rm k} s' \text{ implies } \langle S,\, s\rangle \to s'.$$

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.

$\langle S,\, s\rangle \Rightarrow^{\mathrm{k}} s'$ implies $\langle S,\, s\rangle \rightarrow s'$.

Proof: The proof proceeds by induction on the length of the derivation sequence $\langle S,\, s\rangle \Rightarrow^{\mathrm{k}} s'$, 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 $\mathrm{k} \leq \mathrm{k}_0$ and we shall then prove that it holds for k_0+1. We proceed by cases on how the first step of $\langle S,\, s\rangle \Rightarrow^{\mathrm{k}_0+1} s'$ is obtained, that is by inspecting the derivation tree for the first step of computation in the structural operational semantics.

The case $[\mathrm{ass}_{\mathrm{sos}}]$: Straightforward (and $\mathrm{k}_0 = 0$).

The case $[\mathrm{skip}_{\mathrm{sos}}]$: Straightforward (and $\mathrm{k}_0 = 0$).

The cases $[\mathrm{comp}^1_{\mathrm{sos}}]$ and $[\mathrm{comp}^2_{\mathrm{sos}}]$: In both cases we assume that

$$\langle S_1; S_2,\, s\rangle \Rightarrow^{\mathrm{k}_0+1} s''$$

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

$$\langle S_1,\, s\rangle \Rightarrow^{\mathrm{k}_1} s' \text{ and } \langle S_2,\, s'\rangle \Rightarrow^{\mathrm{k}_2} s''$$

where $\mathrm{k}_1+\mathrm{k}_2=\mathrm{k}_0+1$. The induction hypothesis can now be applied to each of these derivation sequences because $\mathrm{k}_1 \leq \mathrm{k}_0$ and $\mathrm{k}_2 \leq \mathrm{k}_0$. So we get

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

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

Further composites omitted.

Definitions and proofs

- Three approaches to semantics
  - Compositional definitions
  - Natural semantics
  - SOS
- Three proof principles
  - Structural induction
  - Induction on the shape of derivation trees
  - Induction on the length of derivation sequences

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

Extensions of *While*

$$
\begin{array}{rcl}
S & ::= & x := a \mid \texttt{skip} \mid S_1 ; S_2 \\
 & \mid & \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \\
 & \mid & \texttt{while}\ b\ \texttt{do}\ S \\
 & \mid & \texttt{abort} \\
 & \mid & S_1\ \texttt{or}\ S_2 \\
 & \mid & S_1\ \texttt{par}\ S_2
\end{array}
$$

Aborting a computation

Nondeterminism

Parallelism

Adding **abort**

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

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

NS vs. SOS

$$\begin{aligned} & \texttt{abort} \\ \equiv\ & \texttt{skip} \\ \equiv\ & \texttt{while true do skip}\ ? \end{aligned}$$

- A natural semantics may "succeed" with a final state or it may fail to succeed: this could mean both: abortion or nontermination. One could extend the set of final configurations to specifically distinguish "stuck" configurations due to abort.
- In a SOS, looping is reflected by infinite derivation sequences and abnormal termination by finite derivation sequences ending in a stuck configuration.

© Ralf Lämmel, 2009-2010 unless noted otherwise

2010年6月13日星期

Adding nondeterminism

$\texttt{x} := \texttt{1 or } (\texttt{x} := 2;\ \texttt{x} := \texttt{x} + 2)$ evaluates to 1 or 4.

$$[\text{or}^1_{\text{sos}}] \qquad \langle S_1 \texttt{ or } S_2,\, s\rangle \Rightarrow \langle S_1,\, s\rangle$$

$$[\text{or}^2_{\text{sos}}] \qquad \langle S_1 \texttt{ or } S_2,\, s\rangle \Rightarrow \langle S_2,\, s\rangle$$

$$[\text{or}^1_{\text{ns}}] \qquad \frac{\langle S_1,\, s\rangle \to s'}{\langle S_1 \texttt{ or } S_2,\, s\rangle \to s'}$$

$$[\text{or}^2_{\text{ns}}] \qquad \frac{\langle S_2,\, s\rangle \to s'}{\langle S_1 \texttt{ or } S_2,\, s\rangle \to s'}$$

NS vs. SOS

Does the following program terminate?

$(\texttt{while true do skip}) \texttt{ or } (\texttt{x} := 2;\ \texttt{x} := \texttt{x}+2)$

- In a natural semantics, nondeterminism suppresses looping, if possible, that is, the terminating option will be manifested by any transition (derivation tree).
- In a SOS, nondeterminism does not suppress looping, that is, the derivation sequence could commit to a choice that leads an infinite sequence.

Adding parallelism

$\mathtt{x := 1\ par\ (x := 2;\ x := x+2)}$ evaluates to 1, 3, or 4.

Transition relation for SOS:

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

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

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

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

Transition relation for NS:

$$\frac{(S_1, s) \to s', \quad (S_2, s') \to s''}{(S_1 \text{ par } S_2, s) \to s''}$$

$$\frac{(S_2, s) \to s', \quad (S_1, s') \to s''}{(S_1 \text{ par } S_2, s) \to s''}$$

NS vs. SOS

- In a natural semantics, the execution of the immediate constituents is an atomic entity. Hence, we cannot express interleaving of computations.

  $\mathtt{x := 1\ par\ (x := 2;\ x := x+2)}$ evaluates to 1, 4.

- In a SOS, small steps are modeled and hence interleaving is easily expressed.

  $\mathtt{x := 1\ par\ (x := 2;\ x := x+2)}$ evaluates to 1, 3, or 4.

Blocks and procedures

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

Semantics of var declarations

Extension of semantics of statements:

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

Semantics of variable declarations:

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

$$(\varepsilon,\, s) \to_D s$$

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
```


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

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
```

© Ralf Lämmel, 2009-2010 unless noted otherwise
27
2010年6月13日星期

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
```


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

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

© Ralf Lämmel, 2009-2010 unless noted otherwise

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

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

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)

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)

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

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)

© Ralf Lämmel, 2009-2010 unless noted otherwise 28
2010年6月13日星期

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

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

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

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

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

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

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

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

$[\text{block}_{\text{ns}}]$ $\quad \dfrac{\langle D_V,\ s\rangle \to_D s',\ \text{upd}_{\text{P}}(D_P,\ env_P) \vdash \langle S,\ s'\rangle \to s''}{env_P \vdash \langle \texttt{begin}\ D_V\ D_P\ S\ \texttt{end},\ s\rangle \to s''[\text{DV}(D_V) \longmapsto s]}$

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

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

NS with dynamic scope rules using an environment

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

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
```


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

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

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

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

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)**

© Ralf Lämmel, 2009-2010 unless noted otherwise

2010年6月13日星期

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)

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

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)

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

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

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

$$\frac{env'_P \vdash \langle S,\ s\rangle \to s'}{env_P \vdash \langle \texttt{call}\ p,\ s\rangle \to s'} \quad \text{where } env_P\ p = (S,\ env'_P)$$

- Recursive calls

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

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
```


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

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

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

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

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)

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)**

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

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)

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

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

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.

© Ralf Lämmel, 2009-2010 unless noted otherwise
32
2010年6月13日星期