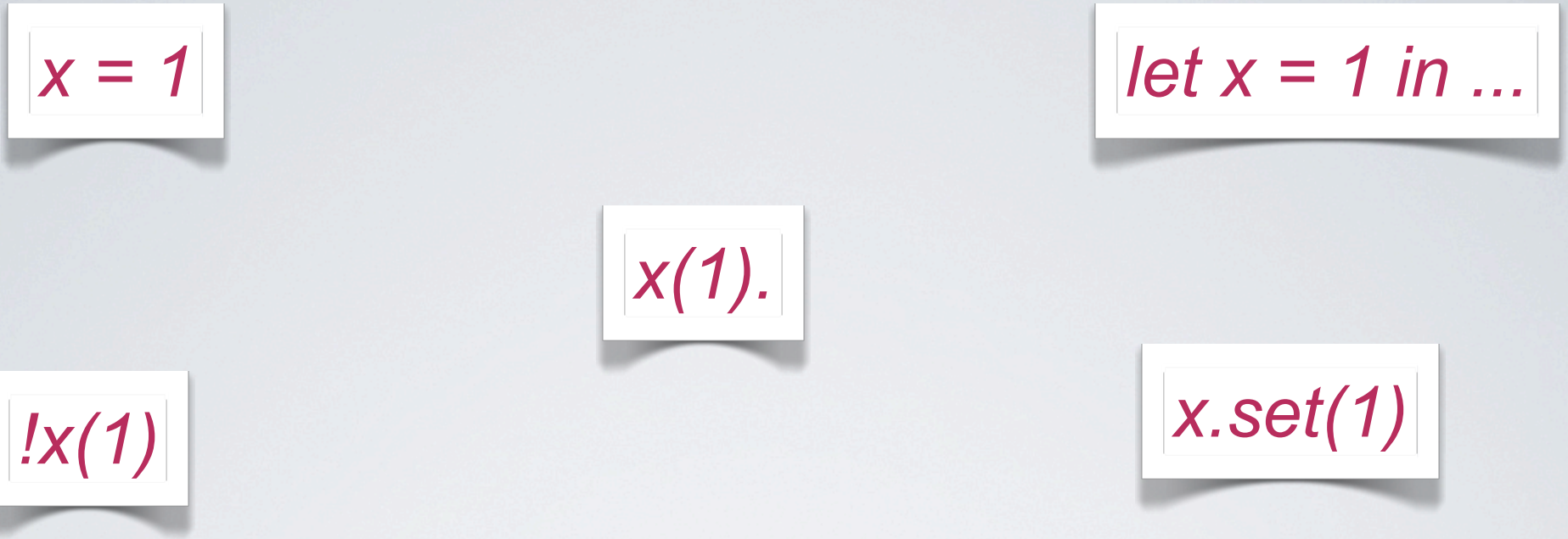

Programming Language Theory

Big-step Operational Semantics (aka Natural Semantics)

Ralf Lämmel

A big-step operational semantics for *While*

Syntactic categories of the *While* language

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

Semantic categories of the *While* language

Natural numbers

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

Truth values

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

States

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

Lookup in a state: $s\ x$

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

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

Meanings of syntactic categories

Numerals

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

Variables

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

Arithmetic expressions

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

Boolean expressions

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

Statements

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

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

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

Semantics of boolean expressions

$$\mathcal{B}[\texttt{true}]s = \mathsf{tt}$$

$$\mathcal{B}[\texttt{false}]s = \mathsf{ff}$$

$$\mathcal{B}[a_1 = a_2]s = \begin{cases} \mathsf{tt} & \text{if } \mathcal{A}[a_1]s = \mathcal{A}[a_2]s \\ \mathsf{ff} & \text{if } \mathcal{A}[a_1]s \neq \mathcal{A}[a_2]s \end{cases}$$

$$\mathcal{B}[a_1 \leq a_2]s = \begin{cases} \mathsf{tt} & \text{if } \mathcal{A}[a_1]s \leq \mathcal{A}[a_2]s \\ \mathsf{ff} & \text{if } \mathcal{A}[a_1]s \not\leq \mathcal{A}[a_2]s \end{cases}$$

$$\mathcal{B}[\neg b]s = \begin{cases} \mathsf{tt} & \text{if } \mathcal{B}[b]s = \mathsf{ff} \\ \mathsf{ff} & \text{if } \mathcal{B}[b]s = \mathsf{tt} \end{cases}$$

$$\mathcal{B}[b_1 \wedge b_2]s = \begin{cases} \mathsf{tt} & \text{if } \mathcal{B}[b_1]s = \mathsf{tt} \\ & \text{and } \mathcal{B}[b_2]s = \mathsf{tt} \\ \mathsf{ff} & \text{if } \mathcal{B}[b_1]s = \mathsf{ff} \\ & \text{or } \mathcal{B}[b_2]s = \mathsf{ff} \end{cases}$$

Semantics of statements

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

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

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

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

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

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

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

Derivation trees

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

Derivation tree

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

States

Prolog as a sandbox for big-step operational semantics

https://slps.svn.sourceforge.net/svnroot/slps/topics/NielsonN07/Prolog/While/NS/

Architecture of the interpreter

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

main.pro

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

% Tests

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

Tests

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

Run tests

```
$ make test
swipl -f main.pro
% library(swi_hooks) compiled into pce_swi_hooks 0.00 sec, 3,992 bytes
%  eval.pro compiled 0.00 sec, 7,008 bytes
%  exec.pro compiled 0.00 sec, 2,872 bytes
%  map.pro compiled 0.00 sec, 1,744 bytes
%  test.pro compiled 0.00 sec, 1,128 bytes
OK: evala(add(num(21),num(21)),_G1199,42)
OK: evala(add(num(21),id(x)),[ (x,21)],42)
OK: exec(while(not(eq(id(x),num(0))),seq(assign(y,mul(id(x),id(y))),assign
(x,sub(id(x),num(1))))),[ (x,5), (y,1)],[ (x,0), (y,120)])
```

Arithmetic expression evaluation

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

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

% Variable reference is evaluated to its current value
evala(id(X),M,Y) :- lookup(M,X,Y).
```

Arithmetic expression evaluation cont'd

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

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

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

Boolean expression evaluation

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

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

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

Skip statement

exec(skip,M,M).

Sequential composition

```
exec(seq(S1,S2),M1,M3) :-
 exec(S1,M1,M2),
 exec(S2,M2,M3).
```

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

Assignment

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

Conditional

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

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

Loop statement

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

% Loop statement with false condition
exec(while(B,_),M,M) :-
 evalb(B,M,ff).
```

Abstract data type for maps (states)

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

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

Test framework

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

Blocks and procedures

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

Semantics of var declarations

Extension of semantics of statements:

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

Semantics of variable declarations:

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

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

Scope rules

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

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

Dynamic scope for variables and procedures

- 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 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
```

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

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

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

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

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

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

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

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

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

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

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

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

- Execution
  - call q
  - **call p (calls outer, say global p)**
  - x := x * 2 (affects inner, say local x)
  - y := x (obviously accesses local x)
- Final value of y = 10

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

Dynamic scope for variables
Static scope for procedures

- Updated environment

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

- 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

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

- Recursive calls

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

Static scope for variables and procedures

- Execution
  - call q
  - call p (calls outer, say global p)
  - **x := x * 2 (affects outer, say global x)**
  - y := x (obviously accesses local x)
- Final value of y = 5

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

Formal semantics omitted here.

- **Summary**: *Big-step operational semantics*
  - *Models relations between syntax, states, values.*
  - *Rule-based modeling (conclusion, premises).*
  - *Computations are derivation trees.*
- **Reading**: *"Semantics with applications"*
  - *Chapter 1, Chapter 2.1, Chapter 2.5*
- **Lab**: *Operational Semantics in Prolog*
- **Outlook**:
  - *Small-step semantics*
  - *Semantics-based reasoning*