Programming Language Semantics and Compiler Design

(Sémantique des Langages de Programmation et Compilation)

Natural Operational Semantics of Languages **Block** and **Proc**

Yliès Falcone

ylies.falcone@univ-grenoble-alpes.fr — www.ylies.fr

Univ. Grenoble Alpes, and LIG-Inria team CORSE

Master of Sciences in Informatics at Grenoble (MoSIG)
Master 1 info

Univ. Grenoble Alpes - UFR IM^{2}AG
www.univ-grenoble-alpes.fr — im2ag.univ-grenoble-alpes.fr

Academic Year 2019 - 2020

Outline - Natural Operational Semantics of Languages **Block** and **Proc**

Outline - Natural Operational Semantics of Languages **Block** and **Proc**

Outline - Natural Operational Semantics of Languages **Block** and **Proc**

Blocks and variable declarations: syntax

Extending language **While** to handle variable declarations.

Definition 1 (Language **Block**)

$$\begin{array}{lll} S & \in & \textbf{Stm} \\ S & ::= & x := a \mid \text{skip} \mid S; S \\ & & \mid \text{if } b \text{ then } S \text{ else } S \text{ fi} \\ & & \mid \text{while } b \text{ do } S \text{ od} \\ & & \mid \textbf{begin } D_V \;\; S \textbf{ end} \end{array}$$

Definition 2 (Syntactic category $\textbf{Dec}_V$)

$$D_V ::= \text{var } \; x; \; D_V \mid \text{var } \; x := a; \; D_V \mid \epsilon$$

Example of program in **Block**

Example 1 (Program in **Block**)

```
begin   var $y$ := 1;
        var $x$ := 1;
             begin   var $x$ := 2;
                         $y := x + 1$
             end;
             $x := y + x$
end
```

Outline - Natural Operational Semantics of Languages **Block** and **Proc**

Introducing Procedures in the syntax

Extending **Block** with procedure declarations.

Definition 3 (Language **Proc**)

- Statements

$$\begin{array}{lll} S & \in & \mathbf{Stm} \\ S & ::= & x := a \mid \text{skip} \mid S_1; S_2 \\ & & \mid \text{if } b \text{ then } S_1 \text{ else } S_2 \text{ fi} \\ & & \mid \text{while } b \text{ do } S \text{ od} \\ & & \mid \mathbf{begin}\ D_V\ D_P\ \ S\ \mathbf{end} \mid \text{call } p \end{array}$$

- Variable declarations:

$$D_V \quad ::= \quad \text{var } x;\ D_V \mid \text{var } x := a;\ D_V \mid \epsilon$$

Definition 4 (Syntactic category $\mathbf{Dec}_P$)

$$D_P ::= \text{proc } p \text{ is } S;\ D_P \mid \epsilon$$

Example: a program with procedures

Example 2 (Program in **Proc**)

```
begin   var  $x := 0$;
        var  $y := 1$;
        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
```

Outline - Natural Operational Semantics of Languages **Block** and **Proc**

Questions about programs in **Block**

Consider the **Block** program in Example 1:

```
begin   var y := 1;
        var x := 1
            begin   var x := 2;
                        y := x + 1
            end;
            x := y + x
end
```

Questions:

1. Are the declarations “active” during declaration execution?
2. Which order to choose when executing the declarations?
3. Do we need to restore the initial state?
4. If so, how to restore the initial state?

Example of program in **Proc**

Consider again the **Proc** program in Example 2:

```
begin   var x := 0;
        var y := 1;
        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
```

The final value of y (and more generally the semantics) depends on the scope of variables and procedures.

Example: a program with procedures

Dynamic scope for variables and procedures

Example 3 (Dynamic scope for variables and procedures)

```
begin   var  $x := 0$;
        var  $y := 1$
        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
```

We need to have some “memorization” of the current “procedure mapping”
$\hookrightarrow$ when we call q we call p and modify x

Example: a program with procedures

Static scope for variables and procedures

Example 4 (Static scope for variables and procedures)

```
begin   var x := 0;
        var y := 1
        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
```

We need to:

- have some "memorization" of the current "procedure mapping" that "remembers the current procedure definitions when it has been defined"

$\hookrightarrow$ when we call q we call p and modify x

Outline - Natural Operational Semantics of Languages **Block** and **Proc**

Outline - Natural Operational Semantics of Languages **Block** and **Proc**

Some preliminary notation: stacks

We use a stack structure to *manage local declarations*.

Let $\mathcal{F}$ be a set of (partial) functions with the same signature.
Elements of $\mathcal{F}$ are denoted by f (which can be subscripted and primed).
We denote by $[\,]$ the empty partial function (defined nowhere, i.e., $\mathtt{Dom}([\,]) = \emptyset$)

Stack notation over partial functions

- The set of stacks over $\mathcal{F}$ is denoted by $\mathcal{F}^*$.
- Elements of $\mathcal{F}^*$ are noted $\hat{f}, \hat{f}_1, \hat{f}_2, \ldots$

Definition 5 (Stack)

Stacks are defined inductively:

- The empty stack is denoted by $\emptyset$.
- Given a stack $\hat{f}$ and a partial function f, $\hat{f} \oplus f$ denotes the stack composed of the stack $\hat{f}$ on top of which is partial function f.

Remark A stack can be seen as a sequence where:

- the *push* operation consists in *appending* to the right,
- the *pop* operation consists in *suppressing* from the right.

□

Remark $\emptyset$ is the neutral element of the push operation: $\emptyset \oplus \hat{f} = \hat{f} \oplus \emptyset = \hat{f}$. □

Some preliminary notation: stacks

Example 5 (Playing around with stacks)

Consider the partial functions:
$f = [x \mapsto 0, y \mapsto 1, z \mapsto 100]$, $f' = [x \mapsto 10, y \mapsto 11]$, $f'' = [x \mapsto 20]$.

Some example stacks (of functions) constructed with the above functions:

- $\hat{f}_1 = f \oplus f'$ is the stack made of 2 functions, where f is the element at the back of the stack and f' the element at the top of the stack,
  $\hat{f}_1$ is obtained by pushing f over $\emptyset$, and then pushing f';
- $\hat{f}_2 = f \oplus f' \oplus f''$ is the stack made of 3 functions, where f'' is the top most element,
  $\hat{f}_2 = \hat{f}_1 \oplus f''$, i.e., $\hat{f}_2$ is obtained by pushing f'' over $\hat{f}_1$.

Remark In the following, when a stack $\hat{f}$ is reduced to one element, we sometimes use notation f instead of $\hat{f}$. □

Question

How do we perform the evaluation of elements with a stack, that is how is determined the value of the elements in the domain of the functions involved in the construction of a stack?

Some preliminary notations: stacks (ctd)

Definition 6 (Evaluation on stacks)

Given an element x in the domain of the partial functions, we have:

- For the empty stack: $\emptyset(x) = \texttt{undef}$.
- For a non-empty stack $\hat{f} \oplus f'$:

$$(\hat{f} \oplus f')(x) = \begin{cases} f'(x) & \text{if } x \in \texttt{Dom}(f'), \\ \hat{f}(x) & \text{otherwise.} \end{cases}$$

($\hat{f} \oplus f'$ is the stack resulting from pushing function f' to stack $\hat{f}$.)

Remark Consider the stack $\hat{f} = \hat{f_1} \oplus \hat{f_2}$, $\hat{f_1}$ is a prefix of $\hat{f}$. □

Example 6 (Evaluation on stacks)

Consider the two previous stacks:

- $\hat{f_1} = [x \mapsto 0, y \mapsto 1, z \mapsto 100] \oplus [x \mapsto 10, y \mapsto 11]$, and
- $\hat{f_2} = \hat{f_1} \oplus [x \mapsto 20]$

We have:

- $\hat{f_1}(x) = [x \mapsto 10, y \mapsto 11](x) = 10$ and $\hat{f_2}(x) = [x \mapsto 20](x) = 20$.
- $\hat{f_1}(y) = [x \mapsto 10, y \mapsto 11](y) = 11$ and $\hat{f_2}(y) = \hat{f_1}(y)$.
- $\hat{f_1}(z) = \hat{f_2}(z) = [x \mapsto 0, y \mapsto 1, z \mapsto 100](z) = 100$.

Some preliminary notations: stacks (ctd)

Definition 7 (Substitution on partial functions)

Given some (partial) function $f : E \to F$, $y \in E$, and $v \in F$, $f[y \mapsto v]$ is the partial function defined as:

$$f[y \mapsto v](x) = \begin{cases} v & \text{if } x = y, \\ f(x) & \text{otherwise.} \end{cases}$$

Example 7 (Substitution on partial functions)

Consider again partial function $f = [x \mapsto 0, y \mapsto 1, z \mapsto 100]$.

- $f[z \mapsto 2] = [x \mapsto 0, y \mapsto 1, z \mapsto 2]$,
- $f[w \mapsto 42] = [x \mapsto 0, y \mapsto 1, z \mapsto 100, w \mapsto 42]$.

Outline - Natural Operational Semantics of Languages **Block** and **Proc**

Semantic domains

States are replaced by a symbol table plus a memory:

- a symbol table associates a memory address to a variable (an identifier);
- a memory associates a value to an address.

Definition 8 (Symbol table: variable environment)

$$\mathbf{Env}_V = \mathbf{Var} \overset{\textit{part.}}{\rightarrow} \mathbf{Loc}$$

ρ denotes an element of $\mathbf{Env}_V$. Thus, $\hat{\rho} \in \mathbf{Env}_V{}^*$ denotes a stack of tables.

Definition 9 (Memory)

$$\mathbf{Store} = \mathbf{Loc} \overset{\textit{part.}}{\rightarrow} \mathbb{Z}$$

σ denotes an element of **Store**.

Intuition: a state (as in the previous chapter) corresponds to $\sigma \circ \hat{\rho}$.

Address allocation

new() is a function that returns a *fresh* memory location.

In the rest of this section, we assume that a global variable environment $\hat{\rho}$ is provided. The next section describes how to effectively construct variable environments using blocks.

Semantic functions for arithmetic and boolean expressions

Def. of Aexp

$$\begin{array}{lll} a & \in & \textbf{Aexp} \\ a & ::= & n \mid x \\ & & \mid a + a \mid a - a \mid a * a \end{array}$$

Previous def. of $\mathcal{A}$

$$\begin{array}{rcl} \mathcal{A}[n]\sigma & = & \mathcal{N}(n) \\ \mathcal{A}[x]\sigma & = & \sigma(x) \\ \mathcal{A}[a_1 @ a_2]\sigma & = & \mathcal{A}[a_1]\sigma @_I \mathcal{A}[a_2]\sigma \end{array}$$

$@ \in \{+_I, -_I, *_I\}$

Definition 10 (Semantic function for arithmetic expressions - revisited)

$$\mathcal{A} : \textbf{Aexp} \rightarrow ((\textbf{Env}_V{}^* \times \textbf{Store}) \rightarrow \mathbb{Z}) \quad \begin{array}{rcl} \mathcal{A}[n](\hat{\rho}, \sigma) & = & \mathcal{N}(n) \\ \mathcal{A}[x](\hat{\rho}, \sigma) & = & \sigma(\hat{\rho}(x)) \\ \mathcal{A}[a_1 + a_2](\hat{\rho}, \sigma) & = & \mathcal{A}[a_1](\hat{\rho}, \sigma) +_I \mathcal{A}[a_2](\hat{\rho}, \sigma) \\ \mathcal{A}[a_1 * a_2](\hat{\rho}, \sigma) & = & \mathcal{A}[a_1](\hat{\rho}, \sigma) *_I \mathcal{A}[a_2](\hat{\rho}, \sigma) \\ \mathcal{A}[a_1 - a_2](\hat{\rho}, \sigma) & = & \mathcal{A}[a_1](\hat{\rho}, \sigma) -_I \mathcal{A}[a_2](\hat{\rho}, \sigma) \end{array}$$

Example 8 (Semantic function for arithmetic expressions)

Suppose $\hat{\rho} = [x \mapsto l_0, y \mapsto l_1] \oplus [z \mapsto l_2]$ and $\sigma = [l_0 \mapsto 1, l_1 \mapsto 2, l_3 \mapsto 3]$.

$$\begin{array}{rl} \mathcal{A}[x + z](\hat{\rho}, \sigma) = & \mathcal{A}[x](\hat{\rho}, \sigma) +_I \mathcal{A}[z](\hat{\rho}, \sigma) \\ & = \sigma(\hat{\rho}(x)) +_I \sigma(\hat{\rho}(z)) \\ & = \sigma(l_0) +_I \sigma(l_3) = \ldots \end{array}$$

Semantic functions for arithmetic and boolean expressions

Def. of Bexp

$$\begin{array}{lll} b & \in & \mathbf{Bexp} \\ b & ::= & \text{true} \mid \text{false} \\ & & \mid a = a \mid a \leq a \mid \neg b \mid b \wedge b \end{array}$$

Previous def. of $\mathcal{B}$

$$\begin{array}{rcl} \mathcal{B}[\text{true}]\sigma & = & \mathbf{tt} \\ \mathcal{B}[\text{false}]\sigma & = & \mathbf{ff} \\ \mathcal{B}[a_1 = a_2]\sigma & = & \mathcal{A}[a_1]\sigma =_I \mathcal{A}[a_2]\sigma \\ \mathcal{B}[a_1 \leq a_2]\sigma & = & \mathcal{A}[a_1]\sigma \leq_I \mathcal{A}[a_2]\sigma \\ \mathcal{B}[\neg b]\sigma & = & \neg_{\mathbb{B}}\mathcal{B}[b]\sigma \\ \mathcal{B}[b_1 \wedge b_2]\sigma & = & \mathcal{B}[b_1]\sigma \wedge_{\mathbb{B}} \mathcal{B}[b_2]\sigma \end{array}$$

Exercise 1 (Semantic function for boolean expressions - revisited)

Give the semantic function for boolean expressions (using symbol table and memory).

Transition rules for assignment, skip, and sequential composition

Definition 11 (Transition system for **While**)

Configurations:

$$(\mathbf{Stm} \times {\mathbf{Env}_V}^* \times \mathbf{Store}) \cup \mathbf{Store}$$

Final configurations: **Store**

Transitions: $(\mathbf{Stm} \times {\mathbf{Env}_V}^* \times \mathbf{Store}) \cup \mathbf{Store}$

- Assignment:

$$\frac{}{(x := a, \hat{\rho}, \sigma) \to \sigma[\hat{\rho}(x) \mapsto \mathcal{A}[a](\hat{\rho}, \sigma)]}$$

- Skip:

$$\frac{}{(\text{skip}, \hat{\rho}, \sigma) \to \sigma}$$

- Sequential composition:

$$\frac{(S_1, \hat{\rho}, \sigma) \to \sigma' \quad (S_2, \hat{\rho}, \sigma') \to \sigma''}{(S_1; S_2, \hat{\rho}, \sigma) \to \sigma''}$$

Transition rules for while and if

Definition 12 (Transition system for **While**)

- While:
  - if $\mathcal{B}[b](\hat{\rho}, \sigma) = \mathbf{tt}$
    $$\frac{(S, \hat{\rho}, \sigma) \rightarrow \sigma' \quad (\text{while } b \text{ do } S \text{ od}, \hat{\rho}, \sigma') \rightarrow \sigma''}{(\text{while } b \text{ do } S \text{ od}, \hat{\rho}, \sigma) \rightarrow \sigma''}$$
  - if $\mathcal{B}[b](\hat{\rho}, \sigma) = \mathbf{ff}$
    $$\frac{}{(\text{while } b \text{ do } S \text{ od}, \hat{\rho}, \sigma) \rightarrow \sigma}$$

Exercise 2 (Semantic rule for conditional statements)

Give the rules for the if ... then ... else ... fi construct.

Outline - Natural Operational Semantics of Languages **Block** and **Proc**

Transition rules for blocks

To define the semantics, we define:

- a transition system for declarations, and
- an extended transition system for statements.

Definition 13 (Transition system for variable declarations)

- Configurations: $(\mathbf{Dec}_V \times \mathbf{Env}_V{}^* \times \mathbf{Env}_V \times \mathbf{Store}) \cup (\mathbf{Env}_V \times \mathbf{Store})$ i.e., of the form $(D_v, \hat{\rho}, \rho', \sigma)$ or (ρ', σ), where:
  - D_V: sequence of declarations
  - $\hat{\rho}$: global symbol table
  - ρ': local symbol table
  - σ: memory
- Final configurations: $\mathbf{Env}_V \times \mathbf{Store}$ (i.e., of the form (ρ', σ))
- Transitions:

$$\rightarrow_D \subseteq (\mathbf{Dec}_V \times \mathbf{Env}_V{}^* \times \mathbf{Env}_V \times \mathbf{Store}) \times (\mathbf{Env}_V \times \mathbf{Store})$$

  i.e., of the form $(D_v, \hat{\rho}, \rho', \sigma) \rightarrow_D (\rho'', \sigma'')$

Example 9 (Elements of the transition system for variable declarations)

- (non-final) Configurations:
  $(\text{var } \; z := 10 + y; , [x \mapsto l] \oplus [x \mapsto l_0], [y \mapsto l_1], [l_0 \mapsto 42, l_1 \mapsto 12])$
- Final configuration: $([y \mapsto l_1, z \mapsto l_2], [l_0 \mapsto 42, l_1 \mapsto 12, l_2 \mapsto 22])$
- Transition: $(\text{var } \; z := 10 + y; , [x \mapsto l] \oplus [x \mapsto l_0], [y \mapsto l_1], [l_0 \mapsto 42, l_1 \mapsto 12]) \rightarrow_D ([y \mapsto l_1, z \mapsto l_2], [l_0 \mapsto 42, l_1 \mapsto 12, l_2 \mapsto 22])$

Transition rules for blocks

Transition system for variable declarations

To define the semantics, we define:

- a transition system for declarations, and
- an extended transition system for statements.

Definition 14 (Transition system for Variable Declarations)

- Transitions given by the transition relation $\rightarrow_D$ (where $l = \text{new}()$):

$$(\epsilon, \hat{\rho}, \rho', \sigma) \rightarrow_D (\rho', \sigma)$$

$$\frac{(D_V, \hat{\rho}, \rho[x \mapsto l], \sigma) \rightarrow_D (\rho', \sigma')}{(\text{var } \ x; \ D_V, \hat{\rho}, \rho, \sigma) \rightarrow_D (\rho', \sigma')}$$

$$\frac{(D_V, \hat{\rho}, \rho[x \mapsto l], \sigma[l \mapsto \mathcal{A}[a](\hat{\rho} \oplus \rho, \sigma)]) \rightarrow_D (\rho', \sigma')}{(\text{var } \ x := a; \ D_V, \hat{\rho}, \rho, \sigma) \rightarrow_D (\rho', \sigma')}$$

(Having $\hat{\rho}$ and ρ used in $\mathcal{A}$ means that both the global and local environments are used to evaluate expressions.)

Transition rules for blocks - with only uninitialised variables

Transition system for variable declarations with only uninitialised variables

If we allow only declarations of the form var x:

$$D_V \quad ::= \quad \text{var } x;\ D_V \mid \epsilon$$

Then, the transition system for declarations can be simplified.

Definition 15 (Transition system for variable declarations (uninit. vars))

- Configurations: $(\mathbf{Dec}_V \times \mathbf{Env}_V) \cup \mathbf{Env}_V$ (i.e., of the form (D_v, ρ) or ρ)
- Final configurations: $\mathbf{Env}_V$ (i.e., of the form ρ)
- Transitions given by the transition relation $\rightarrow_D$ (where $l = \text{new}()$):

$$(\epsilon, \rho') \rightarrow_D \rho' \qquad\qquad \frac{(D_V, \rho[x \mapsto l]) \rightarrow_D \rho'}{(\text{var } x;\ D_V, \rho) \rightarrow_D \rho'}$$

Example 10 (Transition system for variable declarations (uninit. vars))

- (non-final) Configuration: $(\text{var } y; \text{var } z;, [x \mapsto l_0])$
- Final configuration: $[x \mapsto l_0, y \mapsto l_1, z \mapsto l_2]$
- Transitions:
  - $(\text{var } y; \text{var } z;, [x \mapsto l_0]) \rightarrow_D (\text{var } z; [x \mapsto l_0, y \mapsto l_1])$
  - $(\text{var } z;, [x \mapsto l_0, y \mapsto l_1]) \rightarrow_D [x \mapsto l_0, y \mapsto l_1, z \mapsto l_2]$

Transition rules for blocks

Transition system for statements

To define the semantics we define:

- a transition system for declarations, and
- a transition system for statements.

Definition 16 (Natural operational semantics of **Block**)

- Configurations:

$$\textbf{Stm} \times \textbf{Env}_V{}^* \times \textbf{Store} \cup \textbf{Store}$$

- Transitions:

$$\frac{(D_V, \hat{\rho}, [\,], \sigma) \rightarrow_D (\rho_l, \sigma') \quad (S, \hat{\rho} \oplus \rho_l, \sigma') \rightarrow \sigma''}{(\textbf{begin}\ D_V\ \ S\ \textbf{end}, \hat{\rho}, \sigma) \rightarrow \sigma''}$$

- **OR** Transitions (when there is only un-initialised variables)

$$\frac{(D_V, [\,]) \rightarrow_D \rho_l \quad (S, \hat{\rho} \oplus \rho_l, \sigma) \rightarrow \sigma'}{(\textbf{begin}\ D_V\ \ S\ \textbf{end}, \hat{\rho}, \sigma) \rightarrow \sigma'}$$

Execution of one statement of **Block**

Consider the **Block** program from Example 1:

```
begin   var y := 1;
        var x := 1
        begin var x := 2;  y := x + 1 end;
        x := y + x
end
```

Let us note:

- D_{V_0} : var $y := 1$; var $x := 1$
- S_0 : (**begin** var $x := 2$; $y := x + 1$ **end**); $x := y + x$
  - S_{00} : (**begin** var $x := 2$; $y := x + 1$ **end**)
  - S_{01} : $x := y + x$
- D_{V_1} : var $x := 2$;
- S_1 : $y := x + 1$

Let us compute a derivation tree with root $(\textbf{begin}\ D_{V_0}\ \ S_0\ \textbf{end}, \hat{\rho}_0, \sigma_0) \to \sigma_0''$, where $\hat{\rho}_0 = \emptyset, \sigma_0 = [\,]$.

$$\frac{\dots}{(\textbf{begin}\ D_{V_0}\ \ S_0\ \textbf{end}, \hat{\rho}_0, \sigma_0) \to \sigma_0''}$$

Execution of one statement of **Block** (ctd)

Applying rule of block:

$$\frac{(D_{V_0}, \hat{\rho}_0, [\,], \sigma_0) \rightarrow_D (\rho_1, \sigma_1) \quad (S_0, \hat{\rho}_0 \oplus \rho_1, \sigma_1) \rightarrow \sigma_0''}{(\textbf{begin } D_{V_0} \;\; S_0 \textbf{ end}, \hat{\rho}_0, \sigma_0) \rightarrow \sigma_0''}$$

Applying rules of sequential composition and block:

$$\frac{\dfrac{(D_{V_1}, \hat{\rho}_1, [\,], \sigma_1) \rightarrow_D (\rho_2, \sigma_2) \quad (S_1, \hat{\rho}_1 \oplus \rho_2, \sigma_2) \rightarrow \sigma_3}{(S_{00}, \hat{\rho}_0 \oplus \rho_1, \sigma_1) \rightarrow \sigma_3} \qquad (S_{01}, \hat{\rho}_0 \oplus \rho_1, \sigma_3) \rightarrow \sigma_0''}{(S_0, \hat{\rho}_0 \oplus \rho_1, \sigma_1) \rightarrow \sigma_0''}$$

where:

$$\begin{array}{lcl}
\rho_0 & = & \emptyset \\
\rho_1 & = & [y \mapsto l_1, x \mapsto l_2] \\
\sigma_1 & = & [l_1 \mapsto 1, l_2 \mapsto 1] \\
\hat{\rho}_1 & = & \hat{\rho}_0 \oplus \rho_1 = \emptyset \oplus \rho_1 = \rho_1 \\
\rho_2 & = & [x \mapsto l_3] \\
\sigma_2 & = & [l_1 \mapsto 1, l_2 \mapsto 1, l_3 \mapsto 2] \\
\sigma_3 & = & [l_1 \mapsto 3, l_2 \mapsto 1, l_3 \mapsto 2] \\
\sigma_0'' & = & [l_1 \mapsto 3, l_2 \mapsto 4, l_3 \mapsto 2]
\end{array}$$

Outline - Natural Operational Semantics of Languages **Block** and **Proc**

Outline - Natural Operational Semantics of Languages **Block** and **Proc**

Dynamic scope for variables and procedures: remember the intuition

Recall the **Proc** program from Example 2:

```
begin   var $x := 0$;
        var $y := 1$
        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
```

We need to have some "memorization" of the current "procedure mapping"
$\hookrightarrow$ when we call q we call p and modify x

Semantics with dynamic scope for variables and procedures

Semantic domains and updating procedure environments

Procedure names belong to a syntactic category called **Pname**.

$\mathbf{Env}_V$	$=$	$\mathbf{Var} \overset{part.}{\rightarrow} \mathbf{Loc} \ni \rho$	Variable environment
$\mathbf{Store}$	$=$	$\mathbf{Loc} \overset{part.}{\rightarrow} \mathbb{Z} \ni \sigma$	Store
$\mathbf{Env}_P$	$=$	$\mathbf{Pname} \xrightarrow{part.} \mathbf{Stm} \ni \lambda$	Procedure environment

Example 11 (Procedure environment)

- $[p \mapsto x := x+1]$: procedure name p is associated to statement $x := x+1$.
- $[q \mapsto \text{call } p]$: procedure name q is associated to a call to procedure p.

Definition 17 (Function to update a procedure environment)

We define function upd : $\mathbf{Env}_P \times \mathbf{Dec}_P \rightarrow \mathbf{Env}_P$:

- $\text{upd}(\lambda, \epsilon) = \lambda$, and
- $\text{upd}(\lambda, \text{proc } p \text{ is } S; D_P) = \text{upd}(\lambda[p \mapsto S], D_P)$

- declarations are processed from left to right
- priority to the rightmost declaration in case of multiple declarations

Example 12 (Function to update a procedure environment)

$$\begin{aligned} &\text{upd}([\,], \text{proc } p \text{ is } x := x+1; \text{proc } q \text{ is call } p) \\ &= \text{upd}([p \mapsto x := x+1], \text{proc } q \text{ is call } p) \\ &= [p \mapsto x := x+1, q \mapsto \text{call } p] \end{aligned}$$

Semantics with dynamic scope for variables and procedures

Semantic domains and updating procedure environments

Procedure names belong to a syntactic category called **Pname**.

$\mathbf{Env}_V$	$=$	$\mathbf{Var} \overset{part.}{\rightarrow} \mathbf{Loc} \ni \rho$	Variable environment
$\mathbf{Store}$	$=$	$\mathbf{Loc} \overset{part.}{\rightarrow} \mathbb{Z} \ni \sigma$	Store
$\mathbf{Env}_P$	$=$	$\mathbf{Pname} \overset{part.}{\longrightarrow} \mathbf{Stm} \ni \lambda$	Procedure environment

Example 13 (Procedure environment)

- $[p \mapsto x := x + 1]$: procedure name p is associated to statement $x := x + 1$.
- $[q \mapsto \text{call } p]$: procedure name q is associated to a call to procedure p.

Definition 18 (Function to update a procedure environment)

We define function upd : $\mathbf{Env}_P \times \mathbf{Dec}_P \rightarrow \mathbf{Env}_P$:

- $\text{upd}(\lambda, \epsilon) = \lambda$, and
- $\text{upd}(\lambda, \text{proc } p \text{ is } S; D_P) = \text{upd}(\boxed{\lambda[p \mapsto S]}, D_P)$

- declarations are processed from left to right
- priority to the rightmost declaration in case of multiple declarations

Example 14 (Function to update a procedure environment)

$$\begin{aligned}
&\text{upd}([\,], \text{proc } p \text{ is } x := x + 1; \text{proc } q \text{ is call } p)\\
&= \text{upd}([p \mapsto x := x + 1], \text{proc } q \text{ is call } p)\\
&= [p \mapsto x := x + 1, q \mapsto \text{call } p]
\end{aligned}$$

Semantics with dynamic scope: transition system

Configurations: $\underbrace{(\mathbf{Stm} \times \mathbf{Env}_P{}^* \times \mathbf{Env}_V{}^* \times \mathbf{Store})}_{\text{non-final configurations}} \quad \cup \quad \underbrace{\mathbf{Store}}_{\substack{\text{final}\\ \text{configurations}}}$

Transition rules:

$$\frac{(D_V, \hat{\rho}, [\,], \sigma) \rightarrow_D (\rho_l, \sigma') \quad (S, \hat{\lambda} \oplus \mathrm{upd}([\,], D_P), \hat{\rho} \oplus \rho_l, \sigma') \rightarrow \sigma''}{(\mathbf{begin}\ D_V\ D_P\ \ S\ \mathbf{end}, \hat{\lambda}, \hat{\rho}, \sigma) \rightarrow \sigma''}$$

OR (when there is only uninitialised variables):

$$\frac{(D_V, [\,]) \rightarrow_D \rho' \quad (S, \hat{\lambda} \oplus \mathrm{upd}([\,], D_P), \hat{\rho} \oplus \rho', \sigma) \rightarrow \sigma''}{(\mathbf{begin}\ D_V\ D_P\ \ S\ \mathbf{end}, \hat{\lambda}, \hat{\rho}, \sigma) \rightarrow \sigma''}$$

$$\frac{(\hat{\lambda}(p), \hat{\lambda}, \hat{\rho}, \sigma) \rightarrow \sigma'}{(\mathrm{call}\ p, \hat{\lambda}, \hat{\rho}, \sigma) \rightarrow \sigma'}$$

We "load" the code associated with p.

Updating the rule for sequential composition:

$$\frac{(S_1, \hat{\lambda}, \hat{\rho}, \sigma) \rightarrow \sigma' \quad (S_2, \hat{\lambda}, \hat{\rho}, \sigma') \rightarrow \sigma''}{(S_1; S_2, \hat{\lambda}, \hat{\rho}, \sigma) \rightarrow \sigma''}$$

Remark S_1 and S_2 execute within the same environments. □

Similarly, other rules are adapted in a straightforward manner...

Semantics with dynamic scope: transition system

Configurations:

$$\underbrace{(\overbrace{\mathbf{Stm}}^{\text{statements}} \times \overbrace{\mathbf{Env}_P{}^*}^{\substack{\text{global}\\ \text{proc. env.}}} \times \overbrace{\mathbf{Env}_V{}^*}^{\substack{\text{global}\\ \text{var. env.}}} \times \overbrace{\mathbf{Store}}^{\text{memory}})}_{\text{non-final configurations}} \cup \underbrace{\overbrace{\mathbf{Store}}^{\text{memory}}}_{\substack{\text{final}\\ \text{configurations}}}$$

Transition rules:

$$\frac{(D_V, \hat\rho, [\,], \sigma) \to_D (\rho_l, \sigma') \quad (S, \boxed{\hat\lambda \oplus \mathsf{upd}([\,], D_P)}, \boxed{\hat\rho \oplus \rho_l}, \sigma') \to \sigma''}{(\textbf{begin}\ D_V\ D_P\ \ S\ \textbf{end}, \hat\lambda, \hat\rho, \sigma) \to \sigma''}$$

OR (when there is only uninitialised variables):

$$\frac{(D_V, [\,]) \to_D \rho' \quad (S, \boxed{\hat\lambda \oplus \mathsf{upd}([\,], D_P)}, \boxed{\hat\rho \oplus \rho'}, \sigma) \to \sigma''}{(\textbf{begin}\ D_V\ D_P\ \ S\ \textbf{end}, \hat\lambda, \hat\rho, \sigma) \to \sigma''}$$

$$\frac{(\boxed{\hat\lambda(p)}, \hat\lambda, \hat\rho, \sigma) \to \sigma'}{(\mathsf{call}\ p, \hat\lambda, \hat\rho, \sigma) \to \sigma'}$$

We "load" the code associated with p.

Updating the rule for sequential composition:

$$\frac{(S_1, \hat\lambda, \hat\rho, \sigma) \to \sigma' \quad (S_2, \hat\lambda, \hat\rho, \sigma') \to \sigma''}{(S_1; S_2, \hat\lambda, \hat\rho, \sigma) \to \sigma''}$$

Remark S_1 and S_2 execute within the same environments. □

Similarly, other rules are adapted in a straightforward manner. . .

Execution of a **Proc** program with dynamic scope for variables and procedures

Syntax of the example program (Example 15)

Example 15 (Program in Proc)

```
begin
        $D_{V_0}$   [ var x := 1
        $D_{P_0}$   [ proc p is x := x * 2;
        $S_0$       [   call p
end
```

Execution of a **Proc** program with dynamic scope for variables and procedures

Execution of the program in Example 15

We start with:

$$\frac{(D_{V_0}, \emptyset, [\,], [\,]) \rightarrow (?_l, ?_m) \quad (S_0, \text{upd}([\,], D_{P_0}), \emptyset \oplus ?_l, ?_m) \rightarrow ?}{(\textbf{begin } D_{V_0}\ D_{P_0}\ \ S_0 \textbf{ end}, \emptyset, \emptyset, [\,]) \rightarrow ?}$$

Let us compute / complete $(D_{V_0}, \emptyset, [\,], [\,]) \rightarrow_D (?_l, ?_m)$:

$$\frac{(\epsilon, \emptyset, [x \mapsto l_0], [l_0 \mapsto 1]) \rightarrow_D ([x \mapsto l_0], [l_0 \mapsto 1])}{(D_{V_0}, \emptyset, [\,], [\,]) \rightarrow_D ([x \mapsto l_0], [l_0 \mapsto 1])}$$

Let us compute $\text{upd}([\,], D_{P_0})$:

$$\text{upd}([\,], \text{proc } p \text{ is } x := x * 2) = \text{upd}([p \mapsto x := x * 2;], \epsilon) = [p \mapsto x := x * 2]$$

Let us compute / complete $(S_0, \text{upd}([\,], D_{P_0}), \emptyset \oplus [x \mapsto l_0], [l_0 \mapsto 1]) \rightarrow ?$:

$$\frac{\overline{(x := x * 2, [p \mapsto x := x * 2], [x \mapsto l_0], [l_0 \mapsto 1]) \rightarrow [l_0 \mapsto 2]}}{(\text{call } p, [p \mapsto x := x * 2], [x \mapsto l_0], [l_0 \mapsto 1]) \rightarrow [l_0 \mapsto 2]}$$

Finally:

$$(\textbf{begin } D_{V_0}\ D_{P_0}\ \ S_0 \textbf{ end}, \emptyset, \emptyset, [\,]) \rightarrow [l_0 \mapsto 2]$$

Execution of a **Proc** program with dynamic scope for variables and procedures

Execution of the program in Example 15

We start with:

$$\frac{(D_{V_0}, \emptyset, [\,], [\,]) \to (?_l, ?_m) \quad (S_0, \boxed{\text{upd}([\,], D_{P_0})}, \emptyset \oplus ?_l, ?_m) \to \ ?}{(\textbf{begin}\ D_{V_0}\ \ D_{P_0}\ \ S_0\ \textbf{end}, \emptyset, \emptyset, [\,]) \to \ ?}$$

Let us compute / complete $(D_{V_0}, \emptyset, [\,], [\,]) \to_D (?_l, ?_m)$:

$$\frac{(\epsilon, \emptyset, [x \mapsto l_0], [l_0 \mapsto 1]) \to_D ([x \mapsto l_0], [l_0 \mapsto 1])}{(D_{V_0}, \emptyset, [\,], [\,]) \to_D ([x \mapsto l_0], [l_0 \mapsto 1])}$$

Let us compute $\text{upd}([\,], D_{P_0})$:

$$\text{upd}([\,], \text{proc } p \text{ is } x := x * 2) = \text{upd}([p \mapsto x := x * 2], \epsilon) = [p \mapsto x := x * 2]$$

Let us compute / complete $(S_0, \boxed{\text{upd}([\,], D_{P_0})}, \boxed{\emptyset \oplus [x \mapsto l_0]}, [l_0 \mapsto 1]) \to ?$:

$$\frac{\overline{(x := x * 2, [p \mapsto x := x * 2], [x \mapsto l_0], [l_0 \mapsto 1]) \to [l_0 \mapsto 2]}}{(\text{call } p, [p \mapsto x := x * 2], [x \mapsto l_0], [l_0 \mapsto 1]) \to [l_0 \mapsto 2]}$$

Finally:

$$(\textbf{begin}\ D_{V_0}\ \ D_{P_0}\ \ S_0\ \textbf{end}, \emptyset, \emptyset, [\,]) \to [l_0 \mapsto 2]$$

Execution of a **Proc** program with dynamic scope for variables and procedures

Syntax of the example program (Example 2)

Recall the **Proc** program from Example 2 and let us introduce some notation:

```
begin
    D_V0  [ var x := 0;
          [ var y := 1

    D_P0  [ proc p is x := x * 2;
          [ proc q is call p;

    S0    [ begin
          [   D_V1  [  var x := 5;
          [   D_P1  [  proc p is x := x + 1;
          [   S1    [  call q; y := x;
          [ end
end
```

Execution of a **Proc** program with dynamic scope for variables and procedures

Execution of the program in Example 2 - Derivation tree for the **outer** block

$$\dfrac{\gamma_0 \to (\rho_1, \sigma_1) \qquad \dfrac{\gamma_1 \to (\rho_2, \sigma_2) \qquad \overbrace{(S_1, \hat{\lambda}_1 \oplus \lambda_2, \hat{\rho}_1 \oplus \rho_2, \sigma_2) \to \sigma_0''}^{T_1}}{(S_0, \hat{\lambda}_1, \rho_1, \sigma_1) \to \sigma_0''}}{(\textbf{begin}\ D_{V_0} D_{P_0}\ S_0\ \textbf{end}, \hat{\lambda}_0, \hat{\rho}_0, \sigma_0) \to \sigma_0''}$$

where

$$\begin{array}{rcl}
\gamma_0 & = & (D_{V_0}, \hat{\rho}_0, \emptyset, \sigma_0) \\
\hat{\lambda}_0 & = & \lambda_0 = \emptyset \\
\hat{\rho}_0 & = & \rho_0 = \emptyset \\
\sigma_0 & = & \emptyset \\
\gamma_1 & = & (D_{V_1}, \hat{\rho}_1, \emptyset, \sigma_1) \\
\rho_1 & = & [x \mapsto l_1, y \mapsto l_2] \\
\sigma_1 & = & [l_1 \mapsto 0, l_2 \mapsto 1] \\
\hat{\lambda}_1 & = & \lambda_1 = \mathrm{upd}([\,], D_{P_0}) = [p \mapsto x := x * 2, q \mapsto \mathrm{call}\ p] \\
\rho_2 & = & [x \mapsto l_3] \\
\sigma_2 & = & [l_1 \mapsto 0, l_2 \mapsto 1, l_3 \mapsto 5] \\
\lambda_2 & = & [p \mapsto x := x + 1]
\end{array}$$

Execution of a **Proc** program with dynamic scope for variables and procedures

Execution of the program in Example 2 - Derivation tree for the **inner** block

Derivation tree T_1:

$$\dfrac{\dfrac{\dfrac{(x := x+1, \hat{\lambda}_1 \oplus \lambda_2, \hat{\rho}_1 \oplus \rho_2, \sigma_2) \to \sigma_3}{(\text{call } p, \hat{\lambda}_1 \oplus \lambda_2, \hat{\rho}_1 \oplus \rho_2, \sigma_2) \to \sigma_3}}{(\text{call } q, \hat{\lambda}_1 \oplus \lambda_2, \hat{\rho}_1 \oplus \rho_2, \sigma_2) \to \sigma_3} \qquad (y := x, \hat{\lambda}_1 \oplus \lambda_2, \hat{\rho}_1 \oplus \rho_2, \sigma_3) \to \sigma_0''}{(S_1, \hat{\lambda}_1 \oplus \lambda_2, \hat{\rho}_1 \oplus \rho_2, \sigma_2) \to \sigma_0''}$$

where

$$\begin{array}{rcl} \sigma_3 & = & [l_1 \mapsto 0, l_2 \mapsto 1, l_3 \mapsto 6] \\ \sigma_0'' & = & [l_1 \mapsto 0, l_2 \mapsto 6, l_3 \mapsto 6] \end{array}$$

Outline - Natural Operational Semantics of Languages **Block** and **Proc**

Static scope for variables and procedures: remember the intuition

Recall the **Proc** program from Example 2:

```
begin   var  x := 0;
        var  y := 1
        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
```

We need to:

- have some "memorization" of the current "procedure mapping" that "remembers the current procedure definitions when it has been defined"

$\hookrightarrow$ when we call q we call p and modify x

Semantics with static scope for variables and procedures

Semantic domains

Recall that:

$$\begin{array}{lcll} \mathbf{Env}_V & = & \mathbf{Var} \overset{part.}{\to} \mathbf{Loc} \ni \rho & \text{Local variable environment} \\ {\mathbf{Env}_V}^* & = & \text{stacks over } \mathbf{Env}_V \ni \hat{\rho} & \text{Global variable environment} \\ \mathbf{Store} & = & \mathbf{Loc} \overset{part.}{\to} \mathbb{Z} \ni \sigma & \text{Store} \end{array}$$

We now have:

$$\begin{array}{lcll} \mathbf{Env}_P & = & \mathbf{Pname} \xrightarrow{part.} \mathbf{Stm} \times {\mathbf{Env}_P}^* \times {\mathbf{Env}_V}^* \ni \lambda & \text{Local procedure environment} \\ {\mathbf{Env}_P}^* & = & \text{stacks over } \mathbf{Env}_P \ni \hat{\lambda} & \text{Global procedure env.} \end{array}$$

Example 16 (Local procedure environment)

- $\lambda_1 = [p \mapsto (x := 42 + y, \emptyset, [x \mapsto l_0, y \mapsto l_1] \oplus [x \mapsto l_2])]$
- $\lambda_2 = [q \mapsto (y := x + y; \text{call}\ \ p, \hat{\lambda}_1, [y \mapsto l_1] \oplus [x \mapsto l_2])]$

Semantics with static scope for variables and procedures

Definition 19 (Updating the procedure environment)

$$\text{upd} : \underbrace{\mathbf{Env}_P{}^*}_{\substack{\text{global}\\ \text{proc. env.}}} \times \underbrace{\mathbf{Env}_V{}^*}_{\substack{\text{global}\\ \text{var. env.}}} \times \underbrace{\mathbf{Env}_P}_{\substack{\text{current local}\\ \text{proc. env.}}} \times \underbrace{\mathbf{Dec}_P}_{\substack{\text{procedure}\\ \text{declaration}}} \longrightarrow \underbrace{\mathbf{Env}_P}_{\substack{\text{produced}\\ \text{proc. env.}}}$$

- $\text{upd}(\hat{\lambda}_g, \hat{\rho}, \lambda_l, \epsilon) = \lambda_l$, and
- $\text{upd}(\hat{\lambda}_g, \hat{\rho}, \lambda_l, \text{proc } p \text{ is } S; D_P) = \text{upd}(\hat{\lambda}_g, \hat{\rho}, \lambda_l[p \mapsto (S, \hat{\lambda}_g \oplus \lambda_l, \hat{\rho})], D_P)$.

Example 17 (Updating the procedure environment)

- Starting from the empty local proc. env. [ ] and updating it with the declaration proc p is $x := x * 2$; the global proc. env. is empty ($\emptyset$):

$$\begin{aligned}\text{upd}(\emptyset, \hat{\rho}, [\,], \text{proc } p \text{ is } x := x * 2) &= \text{upd}(\emptyset, \hat{\rho}, [p \mapsto (x := x * 2, \emptyset, \hat{\rho})], \epsilon)\\ &= [p \mapsto (x := x * 2, \emptyset, \hat{\rho})]\end{aligned}$$

Semantics with static scope for variables and procedures

Definition 20 (Updating the procedure environment)

$$\text{upd} : \underbrace{\mathbf{Env}_P{}^*}_{\substack{\text{global}\\ \text{proc. env.}}} \times \underbrace{\mathbf{Env}_V{}^*}_{\substack{\text{global}\\ \text{var. env.}}} \times \underbrace{\mathbf{Env}_P}_{\substack{\text{current local}\\ \text{proc. env.}}} \times \underbrace{\mathbf{Dec}_P}_{\substack{\text{procedure}\\ \text{declaration}}} \longrightarrow \underbrace{\mathbf{Env}_P}_{\substack{\text{produced}\\ \text{proc. env.}}}$$

- $\text{upd}(\hat{\lambda}_g, \hat{\rho}, \lambda_l, \epsilon) = \lambda_l$, and
- $\text{upd}(\hat{\lambda}_g, \hat{\rho}, \lambda_l, \text{proc } p \text{ is } S; D_P)$
  $= \text{upd}(\hat{\lambda}_g, \hat{\rho}, \boxed{\lambda_l[p \mapsto (S, \boxed{\hat{\lambda}_g \oplus \lambda_l}, \hat{\rho})]}, D_P)$.

Example 18 (Updating the procedure environment)

- Starting from the empty local proc. env. $[\,]$ and updating it with the declaration $\text{proc } p \text{ is } x := x * 2$; the global proc. env. is empty ($\emptyset$):

$$\begin{aligned}\text{upd}(\emptyset, \hat{\rho}, [\,], \text{proc } p \text{ is } x := x * 2 \quad &= \text{upd}(\emptyset, \hat{\rho}, \boxed{[p \mapsto (x := x * 2, \emptyset, \hat{\rho})]}, \epsilon)\\ &= \boxed{[p \mapsto (x := x * 2, \emptyset, \hat{\rho})]}\end{aligned}$$

Semantics with static scope for variables and procedures: transition system

Configurations:

$$\underbrace{(\overbrace{\textbf{Stm}}^{\text{statements}} \times \overbrace{\textbf{Env}_P{}^*}^{\substack{\text{global}\\\text{proc. env.}}} \times \overbrace{\textbf{Env}_V{}^*}^{\substack{\text{global}\\\text{var. env.}}} \times \overbrace{\textbf{Store}}^{\text{memory}})}_{\text{non-final configurations}} \cup \underbrace{\textbf{Store}}_{\substack{\text{final}\\\text{configurations}}}$$

Transition rules:

- Block:

$$\frac{(D_V, \hat{\rho}, [\,], \sigma) \to_D (\rho_l, \sigma') \quad (S, \hat{\lambda} \oplus \text{upd}(\hat{\lambda}, \hat{\rho} \oplus \rho_l, [\,], D_P), \hat{\rho} \oplus \rho_l, \sigma') \to \sigma''}{(\textbf{begin } D_V \ D_P \ \ S \textbf{ end}, \hat{\lambda}, \hat{\rho}, \sigma) \to \sigma''}$$

OR (when there is only uninitialised variables):

$$\frac{(D_V, [\,]) \to_D \rho_l \quad (S, \hat{\lambda} \oplus \text{upd}(\hat{\lambda}, \hat{\rho} \oplus \rho_l, [\,], D_P), \hat{\rho} \oplus \rho_l, \sigma) \to \sigma'}{(\textbf{begin } D_V \ D_P \ \ S \textbf{ end}, \hat{\lambda}, \hat{\rho}, \sigma) \to \sigma'}$$

- Procedure call:

$$\frac{(S, \hat{\lambda}', \hat{\rho}', \sigma) \to \sigma''}{(\text{call } p, \hat{\lambda}, \hat{\rho}, \sigma) \to \sigma''}$$

where $\hat{\lambda}(p) = (S, \hat{\lambda}', \hat{\rho}')$.

We "load" the code and environments associated with p (memory is not modified).

- The other rules are as in the case of dynamic scope.

Semantics with static scope for variables and procedures: transition system

Configurations:

$$\underbrace{(\underbrace{\overset{\text{statements}}{\mathbf{Stm}}}\times\overset{\text{global}}{\overset{\text{proc. env.}}{\mathbf{Env}_P{}^*}}\times\overset{\text{global}}{\overset{\text{var. env.}}{\mathbf{Env}_V{}^*}}\times\overset{\text{memory}}{\mathbf{Store}})}_{\text{non-final configurations}}\cup\underbrace{\overset{\text{memory}}{\mathbf{Store}}}_{\text{final configurations}}$$

Transition rules:

- Block:

$$\frac{(D_V,\hat\rho,[\,],\sigma)\to_D(\rho_l,\sigma')\quad(S,\boxed{\hat\lambda\oplus\mathsf{upd}(\hat\lambda,\hat\rho\oplus\rho_l,[\,],D_P)},\boxed{\hat\rho\oplus\rho_l},\sigma')\to\sigma''}{(\textbf{begin } D_V\ D_P\ \ S\textbf{ end},\hat\lambda,\hat\rho,\sigma)\to\sigma''}$$

  OR (when there is only uninitialised variables):

$$\frac{(D_V,[\,])\to_D\rho_l\quad(S,\boxed{\hat\lambda\oplus\mathsf{upd}(\hat\lambda,\hat\rho\oplus\rho_l,[\,],D_P)},\boxed{\hat\rho\oplus\rho_l},\sigma)\to\sigma'}{(\textbf{begin } D_V\ D_P\ \ S\textbf{ end},\hat\lambda,\hat\rho,\sigma)\to\sigma'}$$

- Procedure call:

$$\frac{(S,\hat\lambda',\hat\rho',\sigma)\to\sigma''}{(\mathsf{call}\ p,\hat\lambda,\hat\rho,\sigma)\to\sigma''}$$

  where $\hat\lambda(p)=(S,\hat\lambda',\hat\rho')$.

  We "load" the code and environments associated with p (memory is not modified).

- The other rules are as in the case of dynamic scope.

Execution of a **Proc** program with static scope for variables and procedures

Execution of the program in Example 15

We start with:

$$\frac{(D_{V_0}, \emptyset, [\,], [\,]) \rightarrow_D (?_l, ?_m) \quad (S_0, \emptyset \oplus \mathrm{upd}(\emptyset, \emptyset \oplus ?_l, [\,], D_{P_0}), \emptyset \oplus ?_l, ?_m) \rightarrow ?}{(\textbf{begin } D_{V_0}\ D_{P_0}\ \ S_0 \textbf{ end}, \emptyset, \emptyset, [\,]) \rightarrow ?}$$

We compute/complete $(D_{V_0}, \emptyset, [\ ], [\ ]) \rightarrow_D (?_l, ?_m)$ (where D_{V_0} is var $x := 1;$):

$$\frac{(\epsilon, \emptyset, [x \mapsto l_0], [l_0 \mapsto 1]) \rightarrow_D ([x \mapsto l_0], [l_0 \mapsto 1])}{(D_{V_0}, \emptyset, [\,], [\,]) \rightarrow_D ([x \mapsto l_0], [l_0 \mapsto 1])}$$

Let us note $\rho_l = \emptyset \oplus [x \mapsto l_0]$ $(=?_l)$ and compute $\mathrm{upd}(\emptyset, \rho_l, [\,], D_{P_0})$:

$\mathrm{upd}(\emptyset, \rho_l, [\,], \text{proc } p \text{ is } x := x * 2;\,)$ $= \mathrm{upd}(\emptyset, \rho_l, [p \mapsto (x := x * 2, \emptyset, \rho_l)], \epsilon)$
(stack of one element) $= [p \mapsto (x := x * 2, \emptyset, \rho_l)] = \lambda_1$

Let us compute / complete $(S_0, \lambda_l, \rho_l, [l_0 \mapsto 1]) \rightarrow ?$:

$$\frac{\overline{(x := x * 2, \emptyset, [x \mapsto l_0], [l_0 \mapsto 1]) \rightarrow [l_0 \mapsto 2]}}{(\text{call } q, \lambda_1, [x \mapsto l_0], [l_0 \mapsto 1]) \rightarrow [l_0 \mapsto 2]}$$

Finally:

$$(\textbf{begin } D_{V_0}\ D_{P_0}\ \ S_0 \textbf{ end}, \emptyset, \emptyset, [\,]) \rightarrow [l_0 \mapsto 2]$$

Execution of a **Proc** program with static scope for variables and procedures

Execution of the program in Example 15

We start with:

$$\frac{(D_{V_0}, \emptyset, [\,], [\,]) \rightarrow_D (?_l, ?_m) \quad (S_0, \emptyset \oplus \text{upd}(\emptyset, \emptyset \oplus ?_l, [\,], D_{P_0}), \emptyset \oplus ?_l, ?_m) \rightarrow \ ?}{(\textbf{begin } D_{V_0} \ D_{P_0} \ \ S_0 \textbf{ end}, \emptyset, \emptyset, [\,]) \rightarrow \ ?}$$

We compute/complete $(D_{V_0}, \emptyset, [\ ], [\ ]) \rightarrow_D (?_l, ?_m)$ (where D_{V_0} is var $x := 1;$):

$$\frac{(\epsilon, \emptyset, [x \mapsto l_0], [l_0 \mapsto 1]) \rightarrow_D ([x \mapsto l_0], [l_0 \mapsto 1])}{(D_{V_0}, \emptyset, [\ ], [\ ]) \rightarrow_D ([x \mapsto l_0], [l_0 \mapsto 1])}$$

Let us note $\rho_l = \boxed{\emptyset \oplus [x \mapsto l_0]}$ $(=?_l)$ and compute $\text{upd}(\emptyset, \rho_l, [\,], D_{P_0})$:

$$\begin{aligned}\text{upd}(\emptyset, \rho_l, [\,], \text{proc } p \text{ is } x := x * 2) \quad &= \text{upd}(\emptyset, \rho_l, \boxed{[p \mapsto (x := x * 2, \emptyset, \rho_l)]}, \epsilon) \\ &= \boxed{[p \mapsto (x := x * 2, \emptyset, \rho_l)]} = \lambda_1\end{aligned}$$

Let us compute / complete $(S_0, \lambda_1, \rho_l, [l_0 \mapsto 1]) \ \rightarrow \ ?$:

$$\frac{\overline{(x := x * 2, \emptyset, [x \mapsto l_0], [l_0 \mapsto 1]) \rightarrow [l_0 \mapsto 2]}}{(\text{call } q, \lambda_1, [x \mapsto l_0], [l_0 \mapsto 1]) \rightarrow [l_0 \mapsto 2]}$$

Finally:

$$(\textbf{begin } D_{V_0} \ D_{P_0} \ \ S_0 \textbf{ end}, \emptyset, \emptyset, [\,]) \rightarrow [l_0 \mapsto 2]$$

Execution of a **Proc** program with static scope for variables and procedures

Execution of the program in Example 2

Let us now consider the **Proc** program in Example 2.

Compared to the execution with dynamic scopes for variables and procedures, the only things that change are:

- function upd, and
- derivation tree T_1.

Changes to function upd:

$$\begin{array}{lcl}
\hat{\lambda}_0 & = & \emptyset = \lambda_0 \\
\hat{\lambda}_{01} & = & [p \mapsto (x := x * 2, \hat{\lambda}_0, \rho_1)] = \lambda_{01} \\
\hat{\lambda}_1 & = & [p \mapsto (x := x * 2, \hat{\lambda}_0, \rho_1), q \mapsto (\text{call } p, \hat{\lambda}_{01}, \rho_1)] = \lambda_1 \\
\lambda_2 & = & [p \mapsto (x := x + 1, \hat{\lambda}_1, \hat{\rho}_1 \oplus \rho_2)]
\end{array}$$

Execution of a **Proc** program with static scope for variables and procedures

Execution of the program in Example 2 - Derivation tree for the **inner** block

Derivation tree T_1:

$$\dfrac{\dfrac{\dfrac{(x := x * 2, \hat{\lambda}_0, \hat{\rho}_1, \sigma_2) \rightarrow \sigma_3}{(\text{call } p, \hat{\lambda}_{01}, \hat{\rho}_1, \sigma_2) \rightarrow \sigma_3}}{(\text{call } q, \hat{\lambda}_1 \oplus \lambda_2, \hat{\rho}_1 \oplus \rho_2, \sigma_2) \rightarrow \sigma_3} \quad (y := x, \hat{\lambda}_1 \oplus \lambda_2, \hat{\rho}_1 \oplus \rho_2, \sigma_3) \rightarrow \sigma_0''}{(S_0, \hat{\lambda}_1 \oplus \lambda_2, \hat{\rho}_1 \oplus \rho_2, \sigma_2) \rightarrow \sigma_0''}$$

where

$$\begin{array}{lcl} \sigma_3 & = & [l_1 \mapsto 0, l_2 \mapsto 1, l_3 \mapsto 5] \\ \sigma_0'' & = & [l_1 \mapsto 0, l_2 \mapsto 5, l_3 \mapsto 5] \end{array}$$

Outline - Natural Operational Semantics of Languages **Block** and **Proc**

Summary - Natural Operational Semantics of Languages **Block** and **Proc**

Summary of Natural Operational Semantics

Definition of the programming languages **While**, **Block**, and **Proc**:

- Syntax with the definitions of syntactic categories: **Var**, **Stm**, **Dec**$_V$, **Dec**$_P$. **Stm**, **Dec**$_V$, **Dec**$_P$ are defined inductively.
- (Declarative) Semantics for arithmetic and Boolean expressions: $\mathcal{A}$, $\mathcal{B}$.
- (Operational) Semantics for statements: $\mathcal{S}_{\text{ns}}$.
- Determinism of the semantics.
- Termination and looping of programs.
- Semantics **Block** (the semantics of declarations is given by a separate transition system).
- Semantics of **Proc**, giving a semantics to procedure declarations and calls (environment for procedures):
  - dynamic scope for variables and procedures;
  - static scope for variables and procedures (symbol table and a memory);
  - dynamic scope for variables and static scope for procedures;
  - static scope for variables and dynamic scope for procedures;
  - recursive vs non-recursive calls (in the tutorial).