Programming Language Semantics and Compiler Design

(Sémantique des Langages de Programmation et Compilation)

Natural Operational Semantics of Language **Proc**

Frédéric Lang & Laurent Mounier
firstname.lastname@univ-grenoble-alpes.fr
Univ. Grenoble Alpes, Inria,
Laboratoire d'Informatique de Grenoble & Verimag

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

Outline - Natural Operational Semantics of Language **Proc**

Outline - Natural Operational Semantics of Language **Proc**

Procedures

In their simplest form, procedures are entities that associate a name to a statement (encapsulation of code).

Procedures leverage:

- abstraction: put focus on the meaning of a procedure (the "what") rather than its code (the "how")
- composition: build complex functions by assembling simpler ones
- code reuse: factoring code

Procedures make programs shorter, easier to develop, maintain, read, test, and verify. Knowing their semantics precisely is important for code correctness.

We extend the language **While** with procedure definitions (containing local variable declarations) and procedure calls.

Procedure definitions

Definition 1 (Syntactic categories of **Proc**)

The syntactic categories **Var**, **Aexp**, and **Bexp** are kept unchanged.
Stm is extended, and three new syntactic categories are introduced:

- **Pid**: procedure identifiers (written $F, F_0, F_1, \ldots$)
- **Pdef**: procedure definitions (written $D, D_0, D_1, \ldots$)
- **Prog**: programs (written $C, C_0, C_1, \ldots$)

Definition 2 (Syntax of procedure definitions)

The syntax of procedure definitions $D \in$ **Pdef** is as follows:

$$D ::= \text{proc } F \text{ is var } x_1, \ldots, x_n \text{ in } S_F \text{ end}$$

where:

- $F \in$ **Pid** is the procedure identifier.
- $S_F \in$ **Stm** is the procedure statement.
- $x_1, \ldots, x_n \in$ **Var** ($n \geq 0$) are the local variable declarations.
  If $n = 0$ then the keyword var may be omitted.

(Procedure parameters will be introduced later)

Examples of procedure definitions

Example 1 (Increment of x)

$$\text{proc incr\_x is } x := x + 1 \text{ end}$$

The initial value of x comes from the caller's state.
The update on x will be visible in the caller's state.

Example 2 (Swap of x and y)

$$\text{proc swap\_x\_y is var } t \text{ in } t := x; x := y; y := t \text{ end}$$

The initial values of x and y come from the caller's state.
The updates on x and y will be visible in the caller's state.
The value of t will not be visible in the caller's state.

Example 3 (Factorial of x – iterative version)

proc fact_x is var y in
  $y := x; z := 1;$ while $y > 0$ do $z := z * y; y := y - 1$ od
end

The initial value of x comes from the caller's state.
The update on z will be visible in the caller's state.
The value of y will not be visible in the caller's state.

Examples of procedure definitions (continued)

proc incr_x is $x := x + 1$ end

proc swap_x_y is var t in $t := x; x := y; y := t$ end

proc fact_x is var y in
  $y := x; z := 1;$ while $y > 0$ do $z := z * y; y := y - 1$ od
end

Exercise 1

- What is the difference between incr_x and the following procedure:
  proc F_1 is var x in $x := x + 1$ end
- What is the difference between swap_x_y and the following procedure:
  proc F_2 is $t := x; x := y; y := t$ end
- What is the difference between fact_x and the following procedure:
  proc F_3 is $z := 1;$ while $x > 0$ do $z := z * x; x := x - 1$ od end

The scope of a variable is the part of the program where the variable is visible.
It begins at its declaration. We will discuss later where it ends.

Extension of statements: procedure call

Definition 3 (Syntax of statements)

The syntactic category **Stm** is extended with procedure calls as follows:

$$\begin{array}{lll} S & ::= & x := a \mid \text{skip} \mid S_1; S_2 \mid \text{if } b \text{ then } S_1 \text{ else } S_2 \text{ fi} \mid \text{while } b \text{ do } S_0 \text{ od} \\ & \mid & F \quad \textit{– procedure call} \end{array}$$

Procedure call is the only change brought to **Stm**. All other syntactic categories of **While** are kept unchanged.

Examples of procedure definitions containing procedure calls

Example 4 (Factorial of x – recursive version)

```
proc fact_x is var y in
    if x ≤ 1 then
        z := 1
    else
        y := x;
        x := x − 1;
        fact_x;
        x := y;
        z := z ∗ x
    fi
end
```

Remark The definition of recursive procedures is cumbersome due to the absence of parameters. Here, the value of x must be explicitly saved (in the local variable y) before the recursive call and restored upon return. □

Programs

Definition 4 (Syntax of programs)

The syntactic category **Prog** of programs $(C, C', C_0, \ldots)$ is defined as follows:

$$C \;::=\; \text{global } x_1, \ldots, x_n \;\; D_1, \ldots, D_m \;\; \text{in } S$$

where:

- $x_1, \ldots, x_n \in$ **Var** $(n \geq 0)$ are the global variables. Their scope is the entire program. We write **glob** for $\{x_1, \ldots, x_n\}$.
- $D_1, \ldots, D_m \in$ **Pdef** $(m \geq 0)$ are the procedure definitions. We write **proc** for $\{D_1, \ldots, D_m\}$. Distinct procedures must have distinct identifiers.
- S is the main statement to be evaluated.

Example 5 (Program)

The following program initializes x and y and then swaps their values:

```
global x, y
proc swap_x_y is var t in t := x; x := y; y := t end
in x := 1; y := 2; swap_x_y
```


Scope of local variables/Variable binding

In **Proc**, every variable must be declared.

Associating a variable with its declaration is called variable binding.

For local variables, binding depends on where the scope of the variable ends.

There are two options:

- Dynamic variable scope: A local variable declared in a procedure F is visible in all procedures called (transitively) from F.

  With dynamic variable scope, binding must be done during the program execution.

- Static variable scope: A local variable declared in a procedure F is visible in F only.

  With static variable scope, binding can be done at compile-time.

Variable binding: Example

Example 6

```
global x
proc F1 is var x in x := 1; F2 end
proc F2 is x := x + 1 end
in x := 0; F1; F2
```

How do the occurrences of x in F_2 bind?

- Dynamic variable scope: x binds to "var x" when F_2 is called from F_1, and to "global x" when F_2 is called from the main statement.

  Upon termination, the global variable x has value 1.

- Static variable scope: x binds to "global x" as it is not declared locally.

  Upon termination, the global variable x has value 2.

Variable binding in **Proc**

For the semantics of **Proc**, we choose static variable binding.

- Variables are easier to track: favors reading, prone to static analysis (variable initialisation).
- More efficient in practice: binding done statically.
- Same choice made by most languages featuring variable declarations.

In the sequel, we will say that a variable is:

- declared if it can be bound to a declaration, undeclared otherwise
- defined if it has a value, undefined otherwise

A program C of **Proc** will have an undefined semantics (impossibility to derive a final configuration) in the following cases:

- C calls a procedure that does not belong to the set **proc**.
- C contains an undeclared variable.
- C reads an undefined variable (same as **While**).

Outline - Natural Operational Semantics of Language **Proc**

Semantics of **Proc**

Main changes of **Proc** with respect to **While**:

- Evaluation depends on two state components:
  - the global state, mapping the global variables to their value
  - the local state, mapping the local variables to their value
- In statements, an additional transition rule is needed for procedure calls.

We revisit the semantics, by defining the transition relation

$$\rightarrow: (\textbf{Stm} \times 2^{\textbf{Var}} \times \textbf{State} \times \textbf{State}) \times (\textbf{State} \times \textbf{State})$$

Transitions have the form $(S, X, \sigma_l, \sigma_g) \rightarrow (\sigma_l', \sigma_g')$ where:

- X is the current set of local variables.
- σ_l, σ_l' represent the local state before and after evaluation of S. They satisfy $\text{dom}(\sigma_l) \subseteq X, \text{dom}(\sigma_l') \subseteq X$.
- σ_g, σ_g' represent the global state before and after evaluation of S. They satisfy $\text{dom}(\sigma_g) \subseteq \textbf{glob}, \text{dom}(\sigma_g') \subseteq \textbf{glob}$.

Remark The sets **proc** and **glob** are constant and accessible all along the execution. Therefore, they are not components of the transition relation. □

New operations on states: State restriction

Definition 5 (State restriction)

Let $\sigma \in \textbf{State}$, $Y \in 2^{\textbf{Var}}$. The restriction of σ wrt. Y, written $\sigma \,/\, Y$, is the state whose domain is restricted to the elements belonging to Y, defined by:

$$\text{dom}(\sigma \,/\, Y) = \text{dom}(\sigma) \cap Y \wedge (\forall y \in \text{dom}(\sigma \,/\, Y))\ (\sigma \,/\, Y)(y) = \sigma(y)$$

We also write $\sigma \setminus Y$ for $\sigma \,/\, (\text{dom}(\sigma) \setminus Y)$, whose domain is restricted to the elements not belonging to Y.

Exercise 2

Let $\sigma = [x \mapsto 1, y \mapsto 3]$.
Give the values of $\sigma / \{\}$, $\sigma / \{x\}$, $\sigma / \{x, y\}$, $\sigma \setminus \{\}$, $\sigma \setminus \{x\}$, and $\sigma \setminus \{x, y\}$?

$$\begin{array}{rcl@{\qquad}rcl} \sigma / \{\} & = & [] & \sigma \setminus \{\} & = & [x \mapsto 1, y \mapsto 3] \\ \sigma / \{x\} & = & [x \mapsto 1] & \sigma \setminus \{x\} & = & [y \mapsto 3] \\ \sigma / \{x, y\} & = & [x \mapsto 1, y \mapsto 3] & \sigma \setminus \{x, y\} & = & [] \end{array}$$

New operations on states: Generalized state update

Definition 6 (Generalized state update)
Let $\sigma, \sigma' \in \textbf{State}$. The update of σ wrt. σ', written $\sigma \bullet \sigma'$, is defined by:

$$\text{dom}(\sigma \bullet \sigma') = \text{dom}(\sigma) \cup \text{dom}(\sigma') \wedge$$
$$(\forall y \in \text{dom}(\sigma \bullet \sigma'))\ (\sigma \bullet \sigma')(y) = \begin{cases} \sigma'(y) & \text{if } y \in \text{dom}(\sigma') \\ \sigma(y) & \text{otherwise} \end{cases}$$

Remark
- $\sigma[x \mapsto v_a] = \sigma \bullet [x \mapsto v_a]$
- $(\forall \sigma \in \textbf{State}, Y \in 2^{\textbf{Var}})\ \sigma = (\sigma \,/\, Y) \bullet (\sigma \setminus Y) = (\sigma \setminus Y) \bullet (\sigma \,/\, Y)$

□

Exercise 3
Let $\sigma_1 = [w \mapsto 0, x \mapsto 1, y \mapsto 3]$ and $\sigma_2 = [y \mapsto 2, z \mapsto 4]$.
Give the values of $\sigma_1 \bullet \sigma_2$, $(\sigma_1 \setminus \{w\}) \bullet \sigma_2$. and $(\sigma_1 \setminus \{y\}) \bullet \sigma_2$.

$$\begin{array}{rcl} \sigma_1 \bullet \sigma_2 & = & [w \mapsto 0, x \mapsto 1, y \mapsto 2, z \mapsto 4] \\ (\sigma_1 \setminus \{w\}) \bullet \sigma_2 & = & [x \mapsto 1, y \mapsto 2, z \mapsto 4] \\ (\sigma_1 \setminus \{y\}) \bullet \sigma_2 & = & [w \mapsto 0, x \mapsto 1, y \mapsto 2, z \mapsto 4] \end{array}$$

Expression evaluation in a given configuration

Exercise 4
In a configuration of the form $(S, X, \sigma_l, \sigma_g)$, what is the value of x? ($x \in \textbf{Var}$)

The value of x is $\sigma(x)$ for some state σ that must satisfy the following:
1. If x is local ($x \in X$), then $\sigma(x) = \sigma_l(x)$
2. If x is not local but global ($x \in \textbf{glob} \setminus X$), then $\sigma(x) = \sigma_g(x)$

- One would be tempted to propose $\sigma = \sigma_g \bullet \sigma_l$, but this is wrong!
  If x is both local undefined and global defined
  then $(\sigma_g \bullet \sigma_l)(x) = \sigma_g(x)$, whereas $\sigma(x)$ should be undefined.
- Right answer: $\sigma = \sigma_g \setminus X \bullet \sigma_l$
  $\sigma_g \setminus X$ ensures that values for (undefined) local variables cannot be gotten from σ_g.

Semantics of skip and assignment

Definition 7 (Transition rule for skip)
Configurations of the form $(\text{skip}, X, \sigma_l, \sigma_g)$.

$$\frac{}{(\text{skip}, X, \sigma_l, \sigma_g) \rightarrow (\sigma_l, \sigma_g)}$$

Definition 8 (Transition rules for assignment)
Configurations of the form $(x := a, X, \sigma_l, \sigma_g)$.
Let $\sigma_x = [x \mapsto \mathcal{A}[a](\sigma_g \setminus X \bullet \sigma_l)]$

$$\frac{}{(x := a, X, \sigma_l, \sigma_g) \rightarrow (\sigma_l \bullet \sigma_x, \sigma_g)}\ \textit{if } x \in X$$

$$\frac{}{(x := a, X, \sigma_l, \sigma_g) \rightarrow (\sigma_l, \sigma_g \bullet \sigma_x)}\ \textit{if } x \in \textbf{glob} \setminus X$$

Remark If x is undeclared ($x \notin \textbf{glob} \cup X$), no transition can be derived. □

Definition 9 (Alternative transition rule for assignment)
The following rule is equivalent to the previous ones:

$$\frac{}{(x := a, X, \sigma_l, \sigma_g) \rightarrow (\sigma_l \bullet (\sigma_x \,/\, X), \sigma_g \bullet (\sigma_x \setminus X))}\ \textit{if } x \in \textbf{glob} \cup X$$

Semantics of sequential composition and while

Definition 10 (Transition rule for sequential composition)
Configurations of the form $(S_1; S_2, X, \sigma_l, \sigma_g)$.

$$\frac{(S_1, X, \sigma_l, \sigma_g) \rightarrow (\sigma_l', \sigma_g') \quad (S_2, X, \sigma_l', \sigma_g') \rightarrow (\sigma_l'', \sigma_g'')}{(S_1; S_2, X, \sigma_l, \sigma_g) \rightarrow (\sigma_l'', \sigma_g'')}$$

Definition 11 (Transition rules for while)
Configurations of the form $(\text{while } b \text{ do } S \text{ od}, X, \sigma_l, \sigma_g)$.
Let $v_b = \mathcal{B}[b](\sigma_g \setminus X \bullet \sigma_l)$

$$\frac{(S, X, \sigma_l, \sigma_g) \rightarrow (\sigma_l', \sigma_g') \quad (\text{while } b \text{ do } S \text{ od}, X, \sigma_l', \sigma_g') \rightarrow (\sigma_l'', \sigma_g'')}{(\text{while } b \text{ do } S \text{ od}, X, \sigma_l, \sigma_g) \rightarrow (\sigma_l'', \sigma_g'')}\ \textit{if } v_b = \textbf{tt}$$

$$\frac{}{(\text{while } b \text{ do } S \text{ od}, X, \sigma_l, \sigma_g) \rightarrow (\sigma_l, \sigma_g)}\ \textit{if } v_b = \textbf{ff}$$

Semantics of conditional statements

Exercise 5 (Transition rules for conditional statements)

Give the rules for configurations of the form if b then S_1 else S_2 fi.

Let $v_b = \mathcal{B}[b](\sigma_g \setminus X \bullet \sigma_l)$

$$\frac{(S_1, X, \sigma_l, \sigma_g) \to (\sigma_l', \sigma_g')}{(\text{if } b \text{ then } S_1 \text{ else } S_2 \text{ fi}, X, \sigma_l, \sigma_g) \to (\sigma_l', \sigma_g')} \quad \textit{if } v_b = \mathbf{tt}$$

$$\frac{(S_2, X, \sigma_l, \sigma_g) \to (\sigma_l', \sigma_g')}{(\text{if } b \text{ then } S_1 \text{ else } S_2 \text{ fi}, X, \sigma_l, \sigma_g) \to (\sigma_l', \sigma_g')} \quad \textit{if } v_b = \mathbf{ff}$$

Semantics of procedure call

Definition 12 (Transition rule for procedure call)

Configurations of the form $(F, X, \sigma_l, \sigma_g)$ where
proc F is var $x_1, \ldots, x_n$ in S_F end $\in$ **proc**

$$\frac{(S_F, \{x_1, \ldots, x_n\}, [], \sigma_g) \to (\sigma_l', \sigma_g')}{(F, X, \sigma_l, \sigma_g) \to (\sigma_l, \sigma_g')}$$

Remark

- A new configuration is created upon procedure call, using the global state and a new, empty local state [].
- Upon procedure termination, the local state σ_l' of the procedure is dropped and the evaluation continues using the global state σ_g' possibly modified by the procedure call and the local state σ_l of the caller.

□

Transition system for programs

Definition 13 (Semantics of programs)

The semantics $\mathcal{S}_{\text{ns}}[C]$ of the program C defined by:

global $x_1, \ldots, x_n$
$D_1, \ldots, D_m$
in S

is defined as follows:

$$\mathcal{S}_{\text{ns}}[C] = \sigma_g \text{ iff } (S, \emptyset, [], []) \to (\sigma_l, \sigma_g)$$

using **proc** $= \{D_1, \ldots, D_m\}$ and **glob** $= \{x_1, \ldots, x_n\}$ throughout

Remark By construction we shall have $\sigma_l = []$ as S may access only the global variables. □

First example of a program execution in **Proc**

global x, y, z
proc F_1 is var x in $x := 2; F_2; y := x$ end
proc F_2 is $x := x + 1; z := x$ end
in $x := 0; F_1$

$$\dfrac{\dfrac{}{(x := 0, \emptyset, [], []) \to ([], [x \mapsto 0])} \quad \dfrac{\dfrac{\dfrac{}{(x := 2, \{x\}, [], [x \mapsto 0]) \to ([x \mapsto 2], [x \mapsto 0])} \quad T_1}{(x := 2; F_2; y := x, \{x\}, [], [x \mapsto 0]) \to ([x \mapsto 2], [x \mapsto 1, y \mapsto 2, z \mapsto 1])}}{(F_1, \emptyset, [], [x \mapsto 0]) \to ([], [x \mapsto 1, y \mapsto 2, z \mapsto 1])}}{(x := 0; F_1, \emptyset, [], []) \to ([], [x \mapsto 1, y \mapsto 2, z \mapsto 1])}$$

T_1:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{(x := x + 1, \emptyset, [], [x \mapsto 0]) \to ([], [x \mapsto 1])} \quad \dfrac{}{(z := x, \emptyset, [], [x \mapsto 1]) \to ([], [x \mapsto 1, z \mapsto 1])}}{(x := x + 1; z := x, \emptyset, [], [x \mapsto 0]) \to ([], [x \mapsto 1, z \mapsto 1])}}{(F_2, \{x\}, [x \mapsto 2], [x \mapsto 0]) \to ([x \mapsto 2], [x \mapsto 1, z \mapsto 1])} \quad T_2}{(F_2; y := x, \{x\}, [x \mapsto 2], [x \mapsto 0]) \to ([x \mapsto 2], [x \mapsto 1, y \mapsto 2, z \mapsto 1])}$$

T_2:

$$\dfrac{}{(y := x, \{x\}, [x \mapsto 2], [x \mapsto 1, z \mapsto 1]) \to ([x \mapsto 2], [x \mapsto 1, y \mapsto 2, z \mapsto 1])}$$

Second example of a program execution in **Proc**: recursive program

global x, z
proc fact is
  var y in if $x \leq 1$ then $z := 1$ else $y := x; x := x - 1;$ fact; $x := y; z := z * x$ fi end
in $x := 2;$ fact

$$\dfrac{\dfrac{}{(x := 2, \emptyset, [], []) \rightarrow ([], [x \mapsto 2])} \quad \dfrac{\dfrac{T_1}{(\text{if } x \leq 1 \text{ then} \ldots \text{else} \ldots \text{fi}, \{y\}, [], [x \mapsto 2]) \rightarrow ([y \mapsto 2], [x \mapsto 2, z \mapsto 2])}}{(\text{fact}, \emptyset, [], [x \mapsto 2]) \rightarrow ([], [x \mapsto 2, z \mapsto 2])}}{(x := 2; \text{fact}, \emptyset, [], []) \rightarrow ([], [x \mapsto 2, z \mapsto 2])}$$

T_1:

$$\dfrac{\dfrac{}{(y := x, \{y\}, [], [x \mapsto 2]) \rightarrow ([y \mapsto 2], [x \mapsto 2])} \quad T_2}{(y := x; x := x - 1; \text{fact}; x := y; z := z * x, \{y\}, [], [x \mapsto 2]) \rightarrow ([y \mapsto 2], [x \mapsto 2, z \mapsto 2])}$$

T_2:

$$\dfrac{\dfrac{}{(x := x - 1, \{y\}, [y \mapsto 2], [x \mapsto 2]) \rightarrow ([y \mapsto 2], [x \mapsto 1])} \quad T_3}{(x := x - 1; \text{fact}; x := y; z := z * x, \{y\}, [y \mapsto 2], [x \mapsto 2]) \rightarrow ([y \mapsto 2], [x \mapsto 2, z \mapsto 2])}$$

Second example of a program execution in **Proc** (continued)

T_3:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{(z := 1, \{y\}, [], [x \mapsto 1]) \rightarrow ([], [x \mapsto 1, z \mapsto 1])}}{(\text{if } x \leq 1 \text{ then} \ldots \text{else} \ldots \text{fi}, \{y\}, [], [x \mapsto 1]) \rightarrow ([], [x \mapsto 1, z \mapsto 1])}}{(\text{fact}, \{y\}, [y \mapsto 2], [x \mapsto 1]) \rightarrow ([y \mapsto 2], [x \mapsto 1, z \mapsto 1])} \quad T_4}{(\text{fact}; x := y; z := z * x, \{y\}, [y \mapsto 2], [x \mapsto 1]) \rightarrow ([y \mapsto 2], [x \mapsto 2, z \mapsto 2])}$$

T_4:

$$\dfrac{\dfrac{}{(x := y, \{y\}, [y \mapsto 2], [x \mapsto 1, z \mapsto 1]) \rightarrow ([y \mapsto 2], [x \mapsto 2, z \mapsto 1])} \quad T_5}{(x := y; z := z * x, \{y\}, [y \mapsto 2], [x \mapsto 1, z \mapsto 1]) \rightarrow ([y \mapsto 2], [x \mapsto 2, z \mapsto 2])}$$

T_5:

$$\dfrac{}{(z := z * x, \{y\}, [y \mapsto 2], [x \mapsto 2, z \mapsto 1]) \rightarrow ([y \mapsto 2], [x \mapsto 2, z \mapsto 2])}$$

Outline - Natural Operational Semantics of Language **Proc**

Procedure parameters

We add two kinds of parameters to procedures:

- input parameters are used to pass values (arguments) to procedures
- output parameters are used to get values (results) from procedures

In a procedure definition, each parameter takes the form of a variable declaration called formal parameter.

A procedure is called with:

- one expression (actual input parameter) for each formal input parameter
- one variable (actual output parameter) for each formal output parameter

Parameter passing works as follows:

- Upon procedure call, the value of each actual input parameter is assigned to the corresponding formal input parameter.
- Upon termination, the value of each formal output parameter is assigned to the variable passed as actual output parameter on the caller's side.

Syntax of procedures and statements, with parameters

Definition 14 (Syntax of procedures with input and output parameters)

$$D ::= \text{proc } F(y_1, \ldots, y_m, ?z_1, \ldots, ?z_p) \text{ is var } x_1, \ldots, x_n \text{ in } S_F \text{ end}$$

where:

- $y_1, \ldots, y_m$ is an arbitrary number $m \geq 0$ of formal input parameters
- $z_1, \ldots, z_p$ is an arbitrary number $p \geq 0$ of formal output parameters
- all the formal parameters and local variables have distinct identifiers

Definition 15 (Updated syntax of statements)

$$\begin{array}{lll} S & ::= & \ldots & \textit{-- same as before} \\ & | & F(a_1, \ldots, a_m, ?w_1, \ldots, ?w_p) & \textit{-- procedure call} \end{array}$$

where $w_1, \ldots, w_p \in \textbf{Var}$ are distinct variables.

Examples: procedures with parameters

Example 7 (Factorial of x – recursive version with parameters)

proc fact$(x, ?y)$ is if $x \leq 1$ then $y := 1$ else fact$(x-1, ?y); y := y * x$ fi end

Example of call: fact$(2, ?z)$.

Example 8 (Swap with parameters)

proc swap$(x, y, ?xo, ?yo)$ is $xo := y; yo := x$ end

Example of call: swap$(x, y, ?x, ?y)$.

Semantics of procedure call with parameters

Definition 16 (Transition rule for procedure call with parameters)

Configurations of the form $(F(a_1, \ldots, a_m, ?w_1, \ldots, ?w_p), X, \sigma_l, \sigma_g)$, where proc $F(y_1, \ldots, y_m, ?z_1, \ldots, ?z_p)$ is var $x_1, \ldots, x_n$ in S_F end $\in$ **proc**

$$\frac{(S_F, \{x_1, \ldots, x_n, y_1, \ldots, y_m, z_1, \ldots, z_p\}, \sigma_{in}, \sigma_g) \to (\sigma'_l, \sigma'_g)}{(F(a_1, \ldots, a_m, ?w_1, \ldots, ?w_p), X, \sigma_l, \sigma_g) \to (\sigma_l \bullet (\sigma_{out} \,/\, X), \sigma'_g \bullet (\sigma_{out} \setminus X))} \quad \textit{if cond}$$

where:

- $cond = \{w_1, \ldots, w_p\} \subseteq \textbf{glob} \cup X$ (the variables w_i must be declared)
- $\sigma_{in} = [y_1 \mapsto \mathcal{A}[a_1](\sigma_g \setminus X \bullet \sigma_l), \ldots, y_m \mapsto \mathcal{A}[a_m](\sigma_g \setminus X \bullet \sigma_l)]$ is the input local state, mapping each formal input parameter to its value.
- $\sigma_{out} = [w_1 \mapsto \sigma'_l(z_1), \ldots, w_p \mapsto \sigma'_l(z_m)]$ is the output local state, mapping each actual output parameter to the value of the corresponding formal output parameter.

Remark $\sigma_{out} \setminus X$ and $\sigma_{out} \,/\, X$ partition the output local state into updates for the current global and local states, respectively. □

Example of derivation: procedure with parameters

global z
proc fact$(x, ?y)$ is
  if $x \leq 1$ then $y := 1$ else fact$(x-1, ?y); y := y * x$ fi end
in fact$(2, ?z)$

$$\dfrac{\dfrac{\dfrac{T_1 \quad \dfrac{}{(y := y * x, \{x, y\}, [x \mapsto 2, y \mapsto 1], []) \to ([x \mapsto 2, y \mapsto 2)], [])}}{(\text{fact}(x-1, ?y); y := y * x, \{x, y\}, [x \mapsto 2], []) \to ([x \mapsto 2, y \mapsto 2], [])}}{(\text{if } x \leq 1 \text{ then } y := 1 \text{ else fact}(x-1, ?y); y := y * x \text{ fi}, \{x, y\}, [x \mapsto 2], []) \to ([x \mapsto 2, y \mapsto 2], [])}}{(\text{fact}(2, ?z), \emptyset, [], []) \to ([], [z \mapsto 2])}$$

where T_1 is the following tree:

$$\dfrac{\dfrac{\dfrac{}{(y := 1, \{x, y\}, [x \mapsto 1], []) \to ([x \mapsto 1, y \mapsto 1], [])}}{(\text{if } x \leq 1 \text{ then } y := 1 \text{ else fact}(x-1, ?y); y := y * x \text{ fi}, \{x, y\}, [x \mapsto 1], []) \to ([x \mapsto 1, y \mapsto 1], [])}}{(\text{fact}(x-1, ?y), \{x, y\}, [x \mapsto 2], []) \to ([x \mapsto 2, y \mapsto 1], [])}$$

Input-output parameters

For now, parameters are either input parameters or output parameters.

The example swap$(x, y, ?x, ?y)$ suggests it would be convenient for some parameters to be both input and output parameters at the same time, without having to duplicate them.

A solution exists with so-called input-output parameters, which allow function swap to be defined more conveniently using only two parameters:
proc swap$(!?x, !?y)$ is var t in $t := x; x := y; y := t$ end

Exercise 6 (May be in TD)

Propose a transition rule for processes with input-output parameters:

$$\text{proc } F(!?z_1, \dots, !?z_m) \text{ is var } x_1, \dots, x_n \text{ in } S_F \text{ end}$$

where calls have the form $F(!?w_1, \dots, !?w_m)$.
Upon call, $w_1, \dots, w_m$ are assigned to $z_1, \dots, z_m$ like input parameters.
Upon termination, $z_1, \dots, z_m$ are assigned to $w_1, \dots, w_m$ like output params.

Exercise

Exercise 7

1. Are the following statements S_1 and S_2 equivalent for all F_1, F_2?

$$S_1 = \text{if } x < 0 \text{ then } F_1() \text{ fi; if } x \geq 0 \text{ then } F_2() \text{ fi}$$
$$S_2 = \text{if } x < 0 \text{ then } F_1() \text{ else } F_2() \text{ fi}$$

   No. Take proc $F_1()$ is $x := -x$ end and initial state $[x \mapsto -1]$. After executing $F_1()$, the state is $[x \mapsto 1]$. S_1 executes $F_2()$ whereas S_2 does not.

2. Are the following statements S_3 and S_4 equivalent for all F?

$$S_3 = F(?x); F(?y) \qquad S_4 = F(?y); F(?x)$$

   No. Take proc $F(?z)$ is $w := w + 1; z := w$ end and initial state $[w \mapsto 0]$.
   After $F(?x); F(?y)$, we get $[w \mapsto 2, x \mapsto 1, y \mapsto 2]$.
   After $F(?y); F(?x)$, we get $[w \mapsto 2, x \mapsto 2, y \mapsto 1]$.

3. Does the following statement S_5 always terminate for all F?

$$S_5 = \text{while } x \geq 0 \text{ do } F(); x := x - 1 \text{ od}$$

   No. Take proc $F()$ is $x := x + 1$ end and initial state $[x \mapsto 0]$.

Side effects

The above programs are not equivalent because of non-local updates, called side effects.

Side effects may be useful for global resources (files, peripherals, etc.) but for variables, they make it hard to reason about programs!

Exercise 8 (May be in TD)

Define a restriction of **Proc** called **pProc** (for *pure* **Proc**), which requires every variable to be declared locally (global variables are replaced by local variables).
Simplify the semantic relation $\rightarrow$ as much as possible.

Reasoning about programs free of side effects

Reasoning is easier for statements that are free of side effects.

Example 9

1. S_1 and S_2 are equivalent in **pProc**.

$$S_1 = \text{if } x < 0 \text{ then } F_1() \text{ fi; if } x \geq 0 \text{ then } F_2() \text{ fi}$$
$$S_2 = \text{if } x < 0 \text{ then } F_1() \text{ else } F_2() \text{ fi}$$

2. S_3 and S_4 are equivalent in **pProc**.

$$S_3 = F(?x); F(?y) \qquad S_4 = F(?y); F(?x)$$

3. S_5 always terminate in **pProc** (provided $F()$ terminates).

$$S_5 = \text{while } x \geq 0 \text{ do } F(); x := x - 1 \text{ od}$$

Outline - Natural Operational Semantics of Language **Proc**

Summary - Natural Operational Semantics of Language **Proc**

Summary of Natural Operational Semantics

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

- Syntax with the definitions of syntactic categories: **Var**, **Aexp**, **Bexp**, **Stm**, and **Pdef** are defined inductively.
- Declarative or Operational Semantics for arithmetic and Boolean expressions: $\mathcal{A}$, $\mathcal{B}$.
- (Operational) Semantics for statements: $\rightarrow$, $\mathcal{S}_{\mathrm{ns}}$.
- Determinism of the semantics.
- Termination and failure of programs.
- Procedure calls, parameters, side effects

Proc is still a very simple language. Other aspects need to be defined in realistic programming languages, e.g.:

- Functions, return, exceptions, loops with break, etc.
- Data types (basic data types, structured data types, dynamic data structures)
- Classes and objets: inheritance, dynamic binding