# Lecture13-14
# Procedural, Functional Abstractions and Parameter Passing

March 18, 2013

# Control Abstractions: Parameter Passing

- Parameters: Where and Why
- Procedure and Function Invocation: In-depth
- By Value Parameter Passing: FUI
- By Name P. P. (command and expression): FUI
- By Need P. P.: Implementative variant
- By Function, Procedure P. P. (Closure Transmission): Implementative variant
- By reference P.P.: FUI
- By Constant P.P.: FUI
- By Result P.P.: FUI
- By Value-Result P.P.: FUI

# Parameters: Where and Why

- **Where.** Anywhere an abstraction (i.e. generalization) is introduced, the problem of its use in possibly, many different contexts, must be considered;
- The use in a specific context requires an *instantiation* mechanism which intimately connects the abstraction to the context of use;
- **Why.**The connection is realized through the use of formal parameters (with which abstraction is made) and the actual parameters which express the context in which abstraction must be used;
- Then, the instantiation of the abstraction consists in the creation of the bindings which connect each formal parameter to the corresponding actual parameter.;
- The mechanism that creates this connection is called Parameter Passing (or P. Transmission).

# Procedure and Function Invocation: In more depth

- Parameter Passing consists of 3 distinct steps:
  - **Transmission:** It evaluates the arguments (i.e. actual parameters) according to the kind of transmission that has been specified (in the corresponding formal parameter) and results into a list, the transmission list, of denotable or storable values, one for each argument;
  - **Binding:** It makes a binding between each formal parameter and the corresponding value of the transmission list. The effective form of the binding depends on the kind of transmission of the formal parameter;
  - **Return:** It binds back the values, computed by body execution, to the arguments that have been passed by one of various form of by result parameter passing. It results in various modifications of the store.

- Hence, the effective structure of the invoked code consists of:
  - Prologue: it includes code for the steps Binding;
  - Body: it corresponds to the body of the procedure or function
  - Epilogue: It includes code for the step Return

- Then, where is put the code for the step Transmission?

- **Q:** Then, where is put the code for the step Transmission?
- **A:** In the code for invocation, of course.

- Step **Transmission** is formally defined by **function** $\mathcal{T}$ in definitional tables.

**Auxiliary Syntactic Domains**

$$\texttt{A} ::= \texttt{byValue E} \mid \texttt{byName E} \mid \texttt{byReference E} \mid \texttt{byConst E} \mid \texttt{byResult E} \mid \texttt{byValueResult E}$$

**Semantic Functions**

$$\mathcal{M}[\![\texttt{Call I } (\texttt{A}_1 \ldots \texttt{A}_n)]\!]_\rho : \texttt{Store} \longrightarrow \texttt{Store}$$

$$\mathcal{T} : \texttt{A}^n \longrightarrow \texttt{Env} \longrightarrow \texttt{Store} \longrightarrow ((\texttt{Val} \cup \texttt{Den})^n \times \texttt{Store})_\perp$$

$$\mathcal{M}[\![\texttt{Call I } (\texttt{A}_1 \ldots \texttt{A}_n)]\!]_\rho(s) =$$
$$\texttt{Let}\{((v_1 \ldots v_n), s_n) = \mathcal{T}[\![(\texttt{A}_1 \ldots \texttt{A}_n)]\!]_\rho(s),\ \texttt{F}(\texttt{f}) = \rho(\texttt{I})\}$$
$$\texttt{f}(v_1 \ldots v_n)(s_n)$$

# Procedure and Function Invocation: In more depth
## The Structure of General Declaration

- Step **Binding** is formally defined by **function** $\mathcal{B}$
- Step **Return** is formally defined by **function** $\mathcal{R}$
- $\mathcal{D}_E$ is for stressing that the declaration of a procedure is an invariant of the store

**Auxiliary Syntactic Domains**

$$\begin{aligned} \mathtt{P} ::= &\ \mathtt{byValue\ I}\ |\mathtt{byName\ I}\ |\mathtt{byReference\ I} \\ &|\mathtt{byConst\ I}\ |\mathtt{byResult\ I}\ |\mathtt{byValueResult\ I} \end{aligned}$$

**Semantic Functions**

$$\mathcal{D}[\![\mathtt{D}]\!] : \mathtt{Env} \to \mathtt{Store} \to (\mathtt{Env} \times \mathtt{Store})_\perp$$

$$\mathcal{D}_E[\![\mathtt{Proc\ I\ (P_1 \ldots P_n)\ C}]\!] : \mathtt{Env} \to \mathtt{Env}$$

$$\mathcal{D}[\![\mathtt{D}]\!]_\rho = \lambda \mathtt{s}.(\mathcal{D}_E[\![\mathtt{D}]\!]_\rho, \mathtt{s})$$

$$\mathcal{B} : \mathtt{P}^\mathtt{n} \to (\mathtt{Env} \times (\mathtt{Val} \cup \mathtt{Den})^\mathtt{n} \times \mathtt{Store}) \to (\mathtt{Env} \times () \times \mathtt{Store})_\perp$$

$$\mathcal{R} : \mathtt{P}^\mathtt{n} \to \mathtt{Env} \to ((\mathtt{Val} \cup \mathtt{Den})^\mathtt{n} \times \mathtt{Store}) \to \mathtt{Store}_\perp$$

$$\begin{aligned} &\mathcal{D}_E[\![\mathtt{Proc\ I\ (P_1 \ldots P_n)\ C}]\!]_\rho = \\ &\quad \mathtt{Let}\{\mathtt{f} = \lambda(\mathtt{v_1 \ldots v_n}).\lambda \mathtt{s}.\mathtt{s_r} \\ &\qquad\quad \mathtt{where}\{(\rho_\mathtt{n}, \_, \mathtt{s_n}) = \mathcal{B}[\![(\mathtt{P_1 \ldots P_n})]\!](\rho, (\mathtt{v_1 \ldots v_n}), \mathtt{s})\} \\ &\qquad\qquad\qquad \{\mathtt{s_c} = \mathcal{M}[\![\mathtt{C}]\!]_{\rho_\mathtt{n}}(\mathtt{s_n})\} \\ &\qquad\qquad\qquad \{\mathtt{s_r} = \mathcal{R}[\![(\mathtt{P_1 \ldots P_n})]\!]\rho_\mathtt{n}((\mathtt{v_1 \ldots v_n}), \mathtt{s_c})\} \\ &\quad\ \ \mathtt{bind}(\mathtt{I}, \mathtt{F}(\mathtt{f}), \rho) \end{aligned}$$

# By Value Parameter Passing: FUI

| By Value Parameter Passing : $\mathcal{T}, \mathcal{B}, \mathcal{R}$ |
|---|
| **Auxiliaries Semantic Functions** |
| $\mathcal{T}[\![(\mathtt{A}_1 \ldots \mathtt{A}_n)]\!]_\rho(\mathtt{s}) = \mathcal{T}_1[\![\mathtt{A}_1]\!]_\rho \circ \ldots \circ \mathcal{T}_1[\![\mathtt{A}_n]\!]_\rho \; ((), \mathtt{s})$ |
| $\mathcal{T}_1[\![\mathtt{byValue\ AMem(E)}]\!]_\rho((v_1 \ldots v_m), \mathtt{s_m}) =$ |
| $\quad \mathtt{Let}\{(v_{m+1}, \mathtt{s}_{m+1}) =_{\perp_S} \mathcal{E}[\![\mathtt{E}]\!]_\rho(\mathtt{s_m})\}((v_1 \ldots v_m \mathtt{VM}(v_{m+1})), \mathtt{s}_{m+1})$ |
| $\mathcal{B}[\![(\mathtt{P}_1 \ldots \mathtt{P}_n)]\!]_\rho((v_1 \ldots v_n), \mathtt{s}) = (\mathcal{B}_1[\![\mathtt{P}_1]\!] \circ \ldots \circ \mathcal{B}_1[\![\mathtt{P}_n]\!] \circ \downarrow^3_{1-3})(\rho, (v_1 \ldots v_n), \mathtt{s})$ |
| $\mathcal{B}_1[\![\mathtt{byValue\ I_k}]\!](\rho_{k-1}, (v_k \ldots v_n), \mathtt{s}_{k-1}) =$ |
| $\quad \mathtt{Let}\{(\mathtt{l_k}, \mathtt{s}'_k) = \mathtt{allocate}(\mathtt{s}_{k-1})\}$ |
| $\quad\quad \{\mathtt{s_k} = \mathtt{upd}(\mathtt{l_k}, \mathtt{v_k}, \mathtt{s}'_k), \rho_k = \mathtt{bind}(\mathtt{I_k}, \mathtt{l_k}, \rho_{k-1})\}$ |
| $\quad (\rho_k, (v_{k+1} \ldots v_n), \mathtt{s_k})$ |
| $\mathcal{R}[\![(\mathtt{P}_1 \ldots \mathtt{P}_n)]\!]_\rho((v_1 \ldots v_n), \mathtt{s}) = (\mathcal{R}_1[\![\mathtt{P}_1]\!]_\rho \circ \ldots \circ \mathcal{R}_1[\![\mathtt{P}_n]\!]_\rho \circ \downarrow^2_2)((v_1 \ldots v_n), \mathtt{s})$ |
| $\mathcal{R}_1[\![\mathtt{byValue\ I_k}]\!]_\rho((v_k \ldots v_n), \mathtt{s}_{k-1}) =$ |
| $\quad \mathtt{Let}\{\mathtt{s_k} = \mathtt{s}_{k-1}\}((v_{k+1} \ldots v_n), \mathtt{s_k})$ |

- The actual parameter must be an expression whose evaluation must result a storable value: This is stressed by AMem(E)
- Binding creates a mutable value $\mathtt{l_k}$
- Return does nothing

# By Value Parameter Passing: FUI /2

- Use.
  - It is the default parameter passing of almost all languages (Algol 60, Simula, Pascal, C/C++, ML, Ocaml, Ada, C#, Java)
  - It makes a One-way connection: The callee has a copy of the storable value in the caller context
  - It is used in some programming techniques, for passing values to the caller and for using the formal parameter as a mutable for temporary values or for an accumulator
  - No Side Effects in the(store of) caller context

- Implementation
  - Similar to that of the variable declaration with initialization

# By Name Parameter Passing: FUI /1

**By Value Parameter Passing** : $\mathcal{T}, \mathcal{B}, \mathcal{R}$

**Auxiliaries Semantic Functions**

$$\mathcal{T}[\![(\mathtt{A}_1 \ldots \mathtt{A}_n)]\!]_\rho(\mathtt{s}) = \mathcal{T}_1[\![\mathtt{A}_1]\!]_\rho \circ \ldots \circ \mathcal{T}_1[\![\mathtt{A}_n]\!]_\rho \; ((), \mathtt{s})$$

$$\mathcal{T}_1[\![\mathtt{byName\ ACodeC(C)}]\!]_\rho((v_1 \ldots v_m), \mathtt{s}_m) = ((v_1 \ldots v_m \mathtt{Z}(\mathcal{M}[\![\mathtt{C}]\!]_\rho)), \mathtt{s}_{m+1})$$

$$\mathcal{T}_1[\![\mathtt{byName\ ACodeE(E)}]\!]_\rho((v_1 \ldots v_m), \mathtt{s}_m) = ((v_1 \ldots v_m \mathtt{Z}(\mathcal{E}[\![\mathtt{E}]\!]_\rho)), \mathtt{s}_{m+1})$$

$$\mathcal{B}[\![(\mathtt{P}_1 \ldots \mathtt{P}_n)]\!]_\rho((v_1 \ldots v_n), \mathtt{s}) = (\mathcal{B}_1[\![\mathtt{P}_1]\!] \circ \ldots \circ \mathcal{B}_1[\![\mathtt{P}_n]\!] \circ \downarrow^3_{1-3})(\rho, (v_1 \ldots v_n), \mathtt{s})$$

$$\mathcal{B}_1[\![\mathtt{byName\ I}_k]\!](\rho_{k-1}, (v_k \ldots v_n), \mathtt{s}_{k-1}) =$$
$$\quad \mathtt{Let}\{\rho_k = \mathtt{bind}(\mathtt{I}_k, v_k, \rho_{k-1}), \mathtt{s}_k = \mathtt{s}_{k-1}\}$$
$$\qquad (\rho_k, (v_{k+1} \ldots v_n), \mathtt{s}_k)$$

$$\mathcal{R}[\![(\mathtt{P}_1 \ldots \mathtt{P}_n)]\!]_\rho((v_1 \ldots v_n), \mathtt{s}) = (\mathcal{R}_1[\![\mathtt{P}_1]\!]_\rho \circ \ldots \circ \mathcal{R}_1[\![\mathtt{P}_n]\!]_\rho \circ \downarrow^2_2)((v_1 \ldots v_n), \mathtt{s})$$

$$\mathcal{R}_1[\![\mathtt{byName\ I}_k]\!]_\rho((v_k \ldots v_n), \mathtt{s}_{k-1}) =$$
$$\qquad \mathtt{Let}\{\mathtt{s}_k = \mathtt{s}_{k-1}\}((v_{k+1} \ldots v_n), \mathtt{s}_k)$$

**Auxiliaries Semantic Functions**

$\mathtt{Z} : \mathtt{Code} \longrightarrow \mathtt{Den}$ (*injection*)

- The actual parameter must be a Code: This is stressed by `ACodeC(C)` and `ACodeE(E)`
- Binding creates a binding between name of the formal parameter and the Code
- Return does nothing

# By Name Parameter Passing: FUI /2

- The code, passed to the callee, is closed with the bindings (i.e. environment) of the caller;
- The code may be an expression (possibly, an anonymous function, *lambda astrazione*) in the scope of the environment of the caller;
- The code may be a denotable expression that is used from caller/callee for sharing a mutable value
- Hence, $\mathcal{E}[\![]\!]$ must be extended on the expressions $\mathtt{Z}(v)$ that are bound to formals

**Semantic Functions**

$$\mathcal{E}[\![\mathtt{Val(I)}]\!]_\rho(\mathtt{s}) = \begin{cases} \ldots \\ v(\mathtt{s}) & \text{if } \rho(\mathtt{I}) \equiv \mathtt{Z}(v), \\ & \quad \text{for } v \in \mathtt{Store} \rightarrow (\mathtt{Val} \times \mathtt{Store})_\perp \end{cases}$$

$$\mathcal{E}[\![\mathtt{Den(I)}]\!]_\rho(\mathtt{s}) = \begin{cases} \ldots \\ (\mathtt{l}, \mathtt{s}) & \text{if } (\rho(\mathtt{I}) \equiv \mathtt{Z}(\mathtt{l})), \mathtt{l} \in \mathtt{Loc} \end{cases}$$

**Example**

By using by name passing, in Algol 60, a code for the computation of expressions with summation like the one below:

$$z = y + 5 \sum_{n \leq x \leq m} (3x^2 - 5x + 2)$$

introduces a specific function for $\sum$ and invokes it, as an operator, in the expression.

# By Name Parameter Passing: FUI /3

- The code may be an expression for repeated, delayed evaluation
- The code may be a denotable expression for caller/callee value sharing

**Example**

In Algol 60, a code for the computation of expressions with summation, may introduce a specific function for $\sum$ and invokes it, as an operator

$$z := y + 5 \sum_{n \le x \le m} (3x^2 - 5x + 2)$$

```
real procedure sum(expr, i, low, high);
  value low, high;
  real expr;
  integer i, low, high;
begin
  real rtn;
  rtn := 0;
  for i := low step 1 until high do
    rtn := rtn + expr;
  sum := rtn;
end
```

```
z := y + 5 * sum(3 * x * x - 5 * x + 2, x, n, m);
```

# By Name Parameter Passing: FUI /4

- The code may be a simple, or structured, command (possibly, a procedure or a procedural abstraction) in the scope of the environment of the caller;
- Hence, C and $\mathcal{M}[\![\,]\!]$ must be extended on the commands $\mathtt{Z}(v)$ that are bound to formals

**Domini Sintattici**

$\mathtt{C} ::= \ldots\ \mathtt{Exec\ I} \mid \ldots$ (*To execute the code bound* (*to the parameter*))

**Funzioni Semantiche**

$$\mathcal{M}[\![\mathtt{Exec\ I}]\!]_\rho(\mathtt{s}) = \mathtt{Let}\{\mathtt{Z}(v) = \rho(\mathtt{I})\}v(\mathtt{s})$$

**Example**

The structured commands of the language, like `while-do`, can be user defined in the following way: $\mathtt{while\ x} \geq \mathtt{y} + \mathtt{z\ do\ begin\ x} := \mathtt{x} - \mathtt{y}; \mathtt{z} := \mathtt{z} + \mathtt{y\ end}$

```
void procedure while-do(expr, com);
  bool expr;
  void com;
begin
  if expr then begin
      exec com; while-do(expr, com)
      end;
end
```

$\mathtt{while\text{-}do}(\mathtt{x} \geq \mathtt{y} + \mathtt{z},\ \mathtt{begin\ x} := \mathtt{x} - \mathtt{y};\ \mathtt{z} := \mathtt{z} + \mathtt{y\ end});$

# By Name Parameter Passing: FUI /4bis

**Example**

$$\texttt{while-do}(\texttt{x} \geq \texttt{y} + \texttt{z},\ \texttt{begin x} := \texttt{x} - \texttt{y};\ \texttt{z} := \texttt{z} + \texttt{y}\ \texttt{end});$$

In Algol 60, commands must be embodied into a (possibly, parameterless) procedure in order to be passed by name. Hence:

```
procedure while-do(expr, com);
  bool expr; procedure com;
begin
  if expr then begin
      com(); while-do(expr, com)
      end;
end
```

```
procedure B() begin x := x - y; z := z + y end;
  ...
while-do(x ≥ y + z, B);
```

# By Name Parameter Passing: FUI /5

- Use.
  - **Originally** in Algol 60 as default and in Simula, as an alternative to by value
  - But, **no today language**, of widespread use, has the by name parameter passing: heavy to implement, inefficient computations, easy to use but insidious (sharing, aliasing)
  - It has been **abandoned** in favour of by reference (Algol 68, Pascal) by procedure, by function (Pascal), by need (Miranda, Haskell), by value-result and by result (ADA)
  - The **most powerful** kind of parameter passing: The callee receives a code from the caller (through which it can share the context of the caller)
    - **Side effects** (on the store, through the environment of the caller)
    - **Aliasing** (on the environment)
    - **Control Abstractions** can generalize also code instead of only data.
    - **Call Back**. Invoke by passing different code depending on the state in which invocation runs (event programming)
    - It supports **Normal/External Evaluation** in Functional Languages: Computation does not diverge where the function is defined
- Implementation.
  - It introduces a new class of values ($\mathtt{Z}$).
  - **Thunk** is used to implement $\mathtt{Z}$: It is a parameterless closure similar to the procedure B in the example on while-do.

# By Need Parameter Passing/1

- Methodological-Implementative Variant of by Name (Miranda, Haskell)
- It is restricted to expressions only (in non-denotable intepretation)
- The passed expression is bound to the corresponding formal parameter
- But the expression is evaluated only once:
  - The first time that the formal parameter is required for a value, the expression is evaluated in the bindings of the caller
  - The value resulting from such a first evaluation, is used for all the next evaluations of the parameter
- Use
  - It perfectly, copes with **Delay Computation** in Functional Programming, where side effects are forbidden;
  - It supports **Normal/External Evaluation** in Functional Languages in a not expensive way
  - It supports lazy evaluation (when constructors use by need)
- Implementation
  - the expression bound to the parameter is replaced by the value resulting from the evaluation
  - It has a simple implementation, in Functional Languages, by using *graph reductions*
  - Alternatively, by using *memoization* techniques

# By Need Parameter Passing: FUI /2

## Example

What about the program, in particular about the value computed by sum, when: 1) parameter expr is by-need; 2) parameter i is by-need?

```
real procedure sum(expr, i, low, high);
  value low, high;
  real expr;
  integer i, low, high;
begin
  real rtn;
  rtn := 0;
  for i := low step 1 until high do
    rtn := rtn + expr;
  sum := rtn;
end
                    z := y + 5 * sum(3 * x * x - 5 * x + 2, x, n, m);
```

- It is a methodological Variant of by-name: What about Side-effect and Aliasing?
- When they happen, it is due to other mechanisms of the language (consider expressions that have the assignment operator, or pointer operator, &, for mutable values)

# By Procedure/Function Parameter Passing: FUI/1

- Methodological-Implementative Variant of by Name (Algol 60, Pascal, delegates in C#, method references in Java 8)
- code to be passed, must be always encapsulated into an (possibly, parameterless) abstraction
- it is as powerful as by name
- Use
  - Call-Back: Very well expressed
  - Higher-Order: Well expressed. However, it is not Functions as First Class values, since:
    - currying: Hence, non partially evaluated applications
    - Functions as computed values
  - But denotable expressions, like second argument "x" in $\mathtt{sum}(3 * \mathtt{x} * \mathtt{x} - 5 * \mathtt{x} + 2,\ \mathtt{x},\ \mathtt{n},\ \mathtt{m})$, must be used only through non-locals of procedures or functions;
- Implementazione.
  - It implements the *closure*: It is a procedure, or function, together with the bindings of all its non-local identifiers, called the non-local frame;
  - Closures may be nameless functions as in the *Lambda-Abstraction* constructor.
  - The non-local frame is chosen in different way, depending on the language
    - shallow binding: in the caller;
    - deep binding: in the introducer, or declarator.

# By Procedure/Function Parameter Passing: FUI/2

**Example**

In this case, procedure sum has first and second argument by function and by procedure, respectively. To express it, we use keywords `function` and `procedure`.

```
real procedure sum(expr, p, low, high);
  value low, high;
  function expr;
  procedure p;
  real expr;
  integer i, low, high;
begin
  real rtn;
  rtn := 0;
  for i := low step 1 until high do
    begin p(i); rtn := rtn + expr() end
  sum := rtn;
end
```

Moreover, we modify the caller in order to include in it, the following declarations:

```
function myexpr(); begin myexpr:=3*x*x-5*x+2 end;
procedure myx(u); begin value u; integer u; x:=u end;
```

Hence, the statement is:

```
z:=y+5*sum(myexp, myx, n, m);
```

# By Procedure/Function Parameter Passing: FUI/3

- In Caml parameter passing is by-value
- However, since Caml is a Functional Language, it has functions as value.
- Then, it implicitly, has, like all the other Functional Language, by Function Parameter Passing
- However, unlike Haskell, it has neither by-need nor lazy evaluation
- But, both can be emulated by using suitable functions (Apply the same to $\sum$)

## Example

A way to deal with by-need and lazy evaluation in Caml.

```
(* delay expressions: encapsulate each expression "e" that must be passed by need,
  into a function delaye = "fun()->e";
  force evaluation: replace each occurrence of a by need parameter, "x" that must
  be evaluated with "x()" *)

# let rec bottom = fun x -> x+bottom x;;
val bottom : int -> int = <fun>
# let g = fun x y -> x=0;;
val g : int -> 'a -> bool = <fun>
# g 3 (bottom 0);;
Stack overflow during evaluation (looping recursion?).
# let gNS = fun xByNeed yByNeed -> xByNeed()=0;;
val gNS : (unit -> int) -> 'a -> bool = <fun>
# gNS (fun ()->3)(fun()-> bottom 0);;
- : bool = false
```

# By Procedure/Function Parameter Passing: FUI/4

- In Caml parameter passing is by-value
- However, since Caml is a Functional Language, it has functions as value.
- Then, it implicitly, has, like all the other Functional Language, by Function Parameter Passing
- However, unlike Haskell, it has neither by-need nor lazy evaluation
- Apply the transformation delay and force to the rephrasing of $\sum$ in Ocaml

**Example**

Definition of the operator sum in Caml: Note the use of force in the use of argument fExpr

$$z = y + 5 \sum_{n \leq x \leq m} (3x^2 - 5x + 2)$$

```
# let sum = fun fExpr ix inf sup ->
    let res = ref 0 in
        for i = inf to sup do
                ix:=i;
                res:= !res + fExpr();
                done;
        !res;;
```

# By Procedure/Function Parameter Passing: FUI/4

- In Caml parameter passing is by-value
- However, since Caml is a Functional Language, it has functions as value.
- Then, it implicitly, has, like all the other Functional Language, by Function Parameter Passing
- However, unlike Haskell, it has neither by-need nor lazy evaluation
- Apply the transformation delay and force to the rephrasing of $\sum$ in Ocaml

**Example**

Definition of the operator sum in Caml: Note the use of delay in the definition of argument Exp

```
(* in any context, in the scope of sum, we can write: *)
# let z = ref 0 and y = ref 0 and x = ref 0 in
      let exp = fun () -> 3 * !x * !x - 5 * !x + 2 in
            z:= !y + 5 * (sum exp x 10 20); !z;;
- : int = 34760
#
(* and again, in a different context: *)
# let z = ref 0 and x = ref 0 in
      let exp = fun () -> !x * !x * !x + 1 in
            z:= !x + (sum exp x 3 7); !z ;;
- : int = 787
#
```

# By Reference Parameter Passing: FUI/1

| By Reference Parameter Passing : $\mathcal{T}, \mathcal{B}, \mathcal{R}$ |
| --- |
| **Auxiliaries Semantic Functions** <br> $\mathcal{T}_1[\![\texttt{byReference Den(E)}]\!]_\rho((v_1 \ldots v_m), \texttt{s}_m) =$ <br> $\quad \texttt{Let}\{(\texttt{l}, \texttt{s}_{m+1}) =_{\perp_S} \mathcal{E}[\![\texttt{Den(E)}]\!]_\rho(\texttt{s}_m)\}((v_1 \ldots v_m \texttt{l}), \texttt{s}_{m+1})$ <br><br> $\mathcal{B}_1[\![\texttt{byReference I}_k]\!](\rho_{k-1}, (v_k \ldots v_n), \texttt{s}_{k-1}) =$ <br> $\quad \texttt{Let}\{\rho_k = \texttt{bind}(\texttt{I}_k, v_k, \rho_{k-1}), \texttt{s}_k = \texttt{s}_{k-1}\}$ <br> $\quad (\rho_k, (v_{k+1} \ldots v_n), \texttt{s}_k)$ <br><br> $\mathcal{R}_1[\![\texttt{byReference I}_k]\!]_\rho((v_k \ldots v_n), \texttt{s}_{k-1}) =$ <br> $\quad \texttt{Let}\{\texttt{s}_k = \texttt{s}_{k-1}\}((v_{k+1} \ldots v_n), \texttt{s}_k)$ |

- The actual parameter must be a denotable expression whose evaluation must result in a mutable value: This should be already checked by the analyzers in the compiler/interpreter front-end
- Binding creates an alias for such a mutable value. This alias results into:
  - Aliasing, if the actual parameter is a non-local variable of the callee: The mutable value has two different name, in the callee, the parameter and the non-local variable;
  - Sharing, if the actual parameter is not a non-local variable: The caller and the callee have two private accesses.

# By Reference Parameter Passing: FUI/2

- It is in all today Procedural Languages, of widespread use:
  - explicitly: Pascal, ADA, C$\sharp$
  - implicitly, by pointer operator &: C, C++
  - implicitly, by sharing: Java(Objects are reference types with possibly mutable, components)
- Use
  - Two-way Transmission. The Caller and the Callee share the access to a mutable value
  - It is used to pass values from the caller and to obtain back the computed values from the callee.
  - But the two-way connection stays active for the entire execution of the callee: Hence
    - Side Effects
    - Aliasing

    may affect the execution of the callee
- Implementazione.
  - A trivial copy in the environment, of the denotation of the mutable value.
  - This simplicity is the secret of its spread use

# By Reference Parameter Passing: FUI/3

- It is simple to implement but insidious to use

**Example**

```
/* An Aliasing in which the mutable bound to z
 has two name in decrement: *x and z. */

int z=7;
void decrement(int *x){
      while(*x>=z)*x=*x-1;
      }
int main(void){
      decrement(&z);
      }
```

- It is simple to implement but is not prowerful

**Example**

Try do re-phrase in C the program for computing $\sum$

# By Constant Parameter Passing: FUI/1

| By Constant Parameter Passing : $\mathcal{T}, \mathcal{B}, \mathcal{R}$ |
|---|
| **Auxiliaries Semantic Functions** |
| $\mathcal{T}_1[\![\texttt{byConst AConst(E)}]\!]_\rho((v_1 \ldots v_\mathtt{m}), \mathtt{s_m}) =$ $\quad \texttt{Let}\{(v_{\mathtt{m}+1}, \mathtt{s_{m+1}}) =_{\perp_S} \mathcal{E}[\![\mathtt{E}]\!]_\rho(\mathtt{s_m})\}((v_1 \ldots v_\mathtt{m} \texttt{VD}(v_{\mathtt{m}+1})), \mathtt{s_{m+1}})$ |
| $\mathcal{B}_1[\![\texttt{byConst I}_\mathtt{k}]\!](\rho_{\mathtt{k}-1}, (v_\mathtt{k} \ldots v_\mathtt{n}), \mathtt{s_{k-1}}) =$ $\quad \texttt{Let}\{\rho_k = \texttt{bind}(\mathtt{I_k}, v_\mathtt{k}, \rho_{\mathtt{k}-1}), \mathtt{s_k} = \mathtt{s_{k-1}}\}$ $\quad (\rho_\mathtt{k}, (v_{\mathtt{k}+1} \ldots v_\mathtt{n}), \mathtt{s_k})$ |
| $\mathcal{R}_1[\![\texttt{byConst I}_\mathtt{k}]\!]_\rho((v_\mathtt{k} \ldots v_\mathtt{n}), \mathtt{s_{k-1}}) =$ $\quad \texttt{Let}\{\mathtt{s_k} = \mathtt{s_{k-1}}\}((v_{\mathtt{k}+1} \ldots v_\mathtt{n}), \mathtt{s_k})$ |
| **Auxiliaries Functions** |
| $\texttt{VD} : \texttt{Val} \rightarrow \texttt{Den}$ |

- The actual parameter must be a denotable immutable value: This should be already checked by the analyzers in the compiler/interpreter front-end
- Binding creates a binding between formal and value
- Return does nothing

# Trasmissione by Constant/2

- Use
  - It is used in ADA and in a few other Programming Languages
  - One-Way connection Caller-to-Callee. The callee has a copy of the values of the caller
  - Used to pass *read-only* values
  - Guaranteeing the complete separation between the working spaces of the caller and of the callee
  - Correctness: Value Integrity
  - No side effects
- Implementazione.
  - Simple and quite Similar to that of by-reference except for the kind of value that are passed

# By Result Parameter Passing: FUI/1

| **By Result Parameter Passing** : $\mathcal{T}, \mathcal{B}, \mathcal{R}$ |
|---|
| **Auxiliaries Semantic Functions** |
| $\mathcal{T}_1[\![\texttt{byResult Den(E)}]\!]_\rho((v_1 \ldots v_m), \texttt{s}_\texttt{m}) =$ <br> $\quad \texttt{Let}\{(\texttt{l}, \texttt{s}_{\texttt{m+1}}) =_{\perp_S} \mathcal{E}[\![\texttt{Den(E)}]\!]_\rho(\texttt{s}_\texttt{m})\}((v_1 \ldots v_m \texttt{l}), \texttt{s}_{\texttt{m+1}})$ |
| $\mathcal{B}_1[\![\texttt{byResult I}_\texttt{k}]\!](\rho_{\texttt{k}-1}, (v_\texttt{k} \ldots v_\texttt{n}), \texttt{s}_{\texttt{k}-1}) =$ <br> $\texttt{Let}\{(\texttt{l}_\texttt{k}, \texttt{s}'_\texttt{k}) = \texttt{allocate}(\texttt{s}_{\texttt{k}-1})\}$ <br> $\{\texttt{s}_\texttt{k} = \texttt{upd}(\texttt{l}_\texttt{k}, \perp_{\texttt{Mem}}, \texttt{s}'_\texttt{k}), \rho_\texttt{k} = \texttt{bind}(\texttt{I}_\texttt{k}, \texttt{l}_\texttt{k}, \rho_{\texttt{k}-1})\}$ <br> $(\rho_\texttt{k}, (v_{\texttt{k}+1} \ldots v_\texttt{n}), \texttt{s}_\texttt{k})$ |
| $\mathcal{R}_1[\![\texttt{byResult I}_\texttt{k}]\!]_\rho((v_\texttt{k} \ldots v_\texttt{n}), \texttt{s}_{\texttt{k}-1}) =$ <br> $\texttt{Let}\{\texttt{w} = \texttt{look}(\rho(\texttt{I}_\texttt{k}), \texttt{s}_{\texttt{k}-1})\}\{\texttt{s}_\texttt{k} = \texttt{upd}(v_\texttt{k}, \texttt{w}, \texttt{s}_{\texttt{k}-1})\}((v_{\texttt{k}+1} \ldots v_\texttt{n}), \texttt{s}_\texttt{k})$ |

- The actual parameter must be a denotable mutable value: This should be already checked by the analyzers in the compiler/interpreter front-end
- Binding creates a binding between formal and a mutable undefined value
- Return copies the value associated to the mutable of the formal, into the value associated to the mutable of the actual parameter.

# By Result Parameter Passing: FUI/2

- Use.
  - It is used in ADA and in a few other Programming Languages
  - One-Way connection Calle-to-Calleer. The callee receives an address where putting the computed value, and when it ends, it puts the value in that address.
  - Used to pass *write-only* values
  - Guaranteeing the complete separation between the working spaces of the caller and of the callee
  - Correctness: Value Integrity
  - No side effects
- Implementation.
  - A plain implementation of what the denotational semantics does in terms of modifications of environment and store

# By Value-Result Parameter Passing: FUI/1

| By Value – Result Parameter Passing : $\mathcal{T}, \mathcal{B}, \mathcal{R}$ |
|---|
| **Auxiliaries Semantic Functions** |

$$\mathcal{T}_1[\![\texttt{byValueResult Den(E)}]\!]_\rho((v_1 \ldots v_\mathtt{m}), \mathtt{s_m}) = \\ \quad \texttt{Let}\{(\mathtt{l}, \mathtt{s_{m+1}}) =_{\perp_S} \mathcal{E}[\![\texttt{Den(E)}]\!]_\rho(\mathtt{s_m})\}((v_1 \ldots v_\mathtt{m}\,\mathtt{l}), \mathtt{s_{m+1}})$$

$$\mathcal{B}_1[\![\texttt{byValueResult I}_\mathtt{k}]\!](\rho_{\mathtt{k}-1}, (v_\mathtt{k} \ldots v_\mathtt{n}), \mathtt{s}_{\mathtt{k}-1}) = \\ \quad \texttt{Let}\{(\mathtt{l_k}, \mathtt{s'_k}) = \texttt{allocate}(\mathtt{s}_{\mathtt{k}-1}), \mathtt{w} = \texttt{look}(v_\mathtt{k}, s_{\mathtt{k}-1})\} \\ \quad\quad \{\mathtt{s_k} = \texttt{upd}(\mathtt{l_k}, \mathtt{w}, \mathtt{s'_k}), \rho_\mathtt{k} = \texttt{bind}(\mathtt{I_k}, \mathtt{l_k}, \rho_{\mathtt{k}-1})\} \\ \quad (\rho_\mathtt{k}, (v_{\mathtt{k}+1} \ldots v_\mathtt{n}), \mathtt{s_k})$$

$$\mathcal{R}_1[\![\texttt{byValueResult I}_\mathtt{k}]\!]_\rho((v_\mathtt{k} \ldots v_\mathtt{n}), \mathtt{s}_{\mathtt{k}-1}) = \\ \quad \texttt{Let}\{\mathtt{w} = \texttt{look}(\rho(\mathtt{I_k}), \mathtt{s}_{\mathtt{k}-1})\}\{\mathtt{s_k} = \texttt{upd}(v_\mathtt{k}, \mathtt{w}, \mathtt{s}_{\mathtt{k}-1})\}((v_{\mathtt{k}+1} \ldots v_\mathtt{n}), \mathtt{s_k})$$

- The actual parameter must be a denotable, mutable value: This should be already checked by the analyzers in the compiler/interpreter front-end
- Binding creates a binding between formal and a mutable value whose associated value is initialized with a copy of the value associated to the mutable value that has been passed as argument
- Return copies the value associated to the mutable of the formal, into the value associated to the mutable of the actual parameter.

# By Value-Result Parameter Passing: FUI/2

- Use.
  - It is used in ADA and in a few other Programming Languages
  - Two-Way connection. it receives an address where taking an initial value for computation, and where putting the computed value. When the callee ends, it puts the value in such an address.
  - Guaranteeing the complete separation between the working spaces of the caller and of the callee
  - Correctness: Value Integrity
  - No side effects
- Implementation.
  - A plain combination of the implementation of by-value and of by-result

# Excercise

We decide to add an iterator *for*, to the language Ocaml. Ocaml already has an iterator *for* but we want to add an iterator having the same structure and behaviour of *for* of Ansi-C :

(a) Explain:
   1. What is the difference of the two *for* and
   2. How the structure and the behaviour of the new one should be;

(b) Give an abstract syntax and a denotational semantics of the new construct;

(c) Show the implementation, in Ocaml, of the new construct ;

(d) Discuss the mechanisms that have been used to do previous point;

(e) Apply the new construct in rephrasing the code below and comment about its running:

```
int x=0;
int y=0;
for(x=y=10; x+y>x*y;x++){x=10; y=10}
```