Formal Languages and Compilers - Exercises
Lecture 7-8
Scoping and Subprograms

17/04/2012

Outline

Data control

How to provide data to operations and subprograms?

Or, similarly, what is the "environment" of the reference by name?

Two major problems

1. one name can denote different objects (e.g. local variables)
2. one object can be denoted by several names (e.g. passing parameters)

To solve these problems the environments were proposed.

Environment

Binding between the names (Ide) and values:

$$Env : Ide \longrightarrow Loc \cup Val$$

Data control

How to provide data to operations and subprograms?

Or, similarly, what is the "environment" of the reference by name?

Two major problems

1. one name can denote different objects (e.g. local variables)
2. one object can be denoted by several names (e.g. passing parameters)

To solve these problems the environments were proposed.

Environment

Binding between the names (Ide) and values:

$$Env : Ide \longrightarrow Loc \cup Val$$

Data control

How to provide data to operations and subprograms?

Or, similarly, what is the "environment" of the reference by name?

Two major problems

1. one name can denote different objects (e.g. local variables)
2. one object can be denoted by several names (e.g. passing parameters)

To solve these problems the environments were proposed.

Environment

Binding between the names (Ide) and values:

$$Env : Ide \longrightarrow Loc \cup Val$$

Data control

How to provide data to operations and subprograms?

Or, similarly, what is the "environment" of the reference by name?

Two major problems

1. one name can denote different objects (e.g. local variables)
2. one object can be denoted by several names (e.g. passing parameters)

To solve these problems the environments were proposed.

Environment

Binding between the names (Ide) and values:

$$\mathrm{Env} : \mathrm{Ide} \longrightarrow \mathrm{Loc} \cup \mathrm{Val}$$

Environments

Operations in programming language that affect the environment:

1. **Creation of binding** `<name, object>`
   Example: declarations, parameters... in the beginning of execution and when entering the subprograms
2. **Use of the environment**
   Example: reference to the identifier (variables, names of subprograms)
3. **Deactivation the binding**
   Example: when `P` calls `Q`, some bindings of `P` are deactivated
4. **Reactivation the binding**
   Example: when `Q` returns control to `P`
5. **Destruction the binding**
   Example: return from subprogram, the end of execution

Environments

Operations in programming language that affect the environment:

1. **Creation of binding** `<name, object>`
   Example: declarations, parameters... in the beginning of execution and when entering the subprograms
2. **Use of the environment**
   Example: reference to the identifier (variables, names of subprograms)
3. **Deactivation the binding**
   Example: when `P` calls `Q`, some bindings of `P` are deactivated
4. **Reactivation the binding**
   Example: when `Q` returns control to `P`
5. **Destruction the binding**
   Example: return from subprogram, the end of execution

Environments

Operations in programming language that affect the environment:

1. **Creation of binding** `<name, object>`
   Example: declarations, parameters... in the beginning of execution and when entering the subprograms
2. **Use of the environment**
   Example: reference to the identifier (variables, names of subprograms)
3. **Deactivation the binding**
   Example: when `P` calls `Q`, some bindings of `P` are deactivated
4. **Reactivation the binding**
   Example: when `Q` returns control to `P`
5. **Destruction the binding**
   Example: return from subprogram, the end of execution

Environments

Operations in programming language that affect the environment:

1. **Creation of binding** `<name, object>`
   Example: declarations, parameters... in the beginning of execution and when entering the subprograms
2. **Use of the environment**
   Example: reference to the identifier (variables, names of subprograms)
3. **Deactivation the binding**
   Example: when `P` calls `Q`, some bindings of `P` are deactivated
4. **Reactivation the binding**
   Example: when `Q` returns control to `P`
5. **Destruction the binding**
   Example: return from subprogram, the end of execution

Environments

Operations in programming language that affect the environment:

1. **Creation of binding** `<name, object>`
   Example: declarations, parameters... in the beginning of execution and when entering the subprograms
2. **Use of the environment**
   Example: reference to the identifier (variables, names of subprograms)
3. **Deactivation the binding**
   Example: when `P` calls `Q`, some bindings of `P` are deactivated
4. **Reactivation the binding**
   Example: when `Q` returns control to `P`
5. **Destruction the binding**
   Example: return from subprogram, the end of execution

Blocks and local variables

Blocks

A block consists of local declarations and commands:

```
begin
    D   ->  local declarations
    C   ->  commands
end
```

Example

A block is like a procedure without parameters

```
x := 5;
{
    int x;
    x:=7;
    printf("%d", x);   -> 7
}
printf("%d", x);       -> 5
```

Outline

Scoping

"Scoping" solves the problem of determining

- when a particular binding <name, object> is active
- which bindings are valid in a particular moment of executio
- which is the environment?

Different environments

LE: local environment. All bindings created/activated in a block/subprogram

NLE: non-local environment. All bindings used (active) but not local

GE: global environment. All bindings shared by all blocks/subprograms. *GE* can be considered:

- as a subset of *NLE*
- separately from *NLE*

Scoping

"Scoping" solves the problem of determining

- when a particular binding <name, object> is active
- which bindings are valid in a particular moment of executio
- which is the environment?

Different environments

LE: local environment. All bindings created/activated in a block/subprogram

NLE: non-local environment. All bindings used (active) but not local

GE: global environment. All bindings shared by all blocks/subprograms. *GE* can be considered:

- as a subset of *NLE*
- separately from *NLE*

Scoping

"Scoping" solves the problem of determining

- when a particular binding <name, object> is active
- which bindings are valid in a particular moment of executio
- which is the environment?

Different environments

LE: local environment. All bindings created/activated in a block/subprogram

NLE: non-local environment. All bindings used (active) but not local

GE: global environment. All bindings shared by all blocks/subprograms. *GE* can be considered:

- as a subset of *NLE*
- separately from *NLE*

Scoping

"Scoping" solves the problem of determining

- when a particular binding <name, object> is active
- which bindings are valid in a particular moment of executio
- which is the environment?

Different environments

LE: local environment. All bindings created/activated in a block/subprogram

NLE: non-local environment. All bindings used (active) but not local

GE: global environment. All bindings shared by all blocks/subprograms. *GE* can be considered:

- as a subset of *NLE*
- separately from *NLE*

Global Environment (*GE*)

Example (C)

```
int a[20];
float b[5];
struct { int i; char n[10]; } c, d;
...
int main() {...}
```

- Contains also all the identifiers (constants, functions, etc.) predefined in the language
- Common table for all the subprograms (including main)
- Concrete implementation:
  - treat *GE* as a record
  - names are compiled as fields of the record
  - in the code, it's sufficient to know the base address of *GE*

Global Environment (*GE*)

Example (C)

```
int a[20];
float b[5];
struct { int i; char n[10]; } c, d;
...
int main() {...}
```

- Contains also all the identifiers (constants, functions, etc.) predefined in the language
- Common table for all the subprograms (including main)
- Concrete implementation:
  - treat *GE* as a record
  - names are compiled as fields of the record
  - in the code, it's sufficient to know the base address of *GE*

Local Environment (LE) - 1

Notation

- P $\Downarrow$ Q: procedure P calls Q
- P $\Uparrow$ Q: procedure P terminates and returns the control to the caller Q

Let's consider the computation

$$\mathtt{P} \Downarrow \mathtt{Q} \Downarrow \mathtt{R} \Uparrow \mathtt{Q} \Uparrow \mathtt{P}$$

what happens to the local environment of Q?

The simple way

Q $\Downarrow$ R when control is passed to R, LE becomes deactivated

R $\Uparrow$ Q when control is passed back to Q, its LE become reactivated

Local Environment (LE) - 1

Notation

- $\mathtt{P} \Downarrow \mathtt{Q}$: procedure P calls Q
- $\mathtt{P} \Uparrow \mathtt{Q}$: procedure P terminates and returns the control to the caller Q

Let's consider the computation

$$\mathtt{P} \Downarrow \mathtt{Q} \Downarrow \mathtt{R} \Uparrow \mathtt{Q} \Uparrow \mathtt{P}$$

what happens to the local environment of Q?

The simple way

$\mathtt{Q} \Downarrow \mathtt{R}$ when control is passed to R, LE becomes deactivated

$\mathtt{R} \Uparrow \mathtt{Q}$ when control is passed back to Q, its LE become reactivated

Local Environment (*LE*) - 2

The management of environment in Q

$$\mathtt{P} \Downarrow \mathtt{Q} \text{ and } \mathtt{Q} \Uparrow \mathtt{P}$$

is more delicate.

DLE: Dynamic Local Environment

- P $\Downarrow$ Q *LE* of Q is created
- Q $\Uparrow$ P *LE* of Q is destroyed

SLE: Static Local Environment

- P $\Downarrow$ Q *LE* of Q is reactivated
- Q $\Uparrow$ P *LE* of Q is deactivated

Local Environment (*LE*) - 2

The management of environment in Q

$$\mathtt{P} \Downarrow \mathtt{Q} \text{ and } \mathtt{Q} \Uparrow \mathtt{P}$$

is more delicate.

DLE: Dynamic Local Environment

- P $\Downarrow$ Q *LE* of Q is created
- Q $\Uparrow$ P *LE* of Q is destroyed

SLE: Static Local Environment

- P $\Downarrow$ Q *LE* of Q is reactivated
- Q $\Uparrow$ P *LE* of Q is deactivated

Local Environment (*LE*) - 2

The management of environment in Q

$$\mathtt{P} \Downarrow \mathtt{Q} \text{ and } \mathtt{Q} \Uparrow \mathtt{P}$$

is more delicate.

DLE: Dynamic Local Environment

P ⇓ Q *LE* of Q is created

Q ⇑ P *LE* of Q is destroyed

SLE: Static Local Environment

P ⇓ Q *LE* of Q is reactivated

Q ⇑ P *LE* of Q is deactivated

Outline

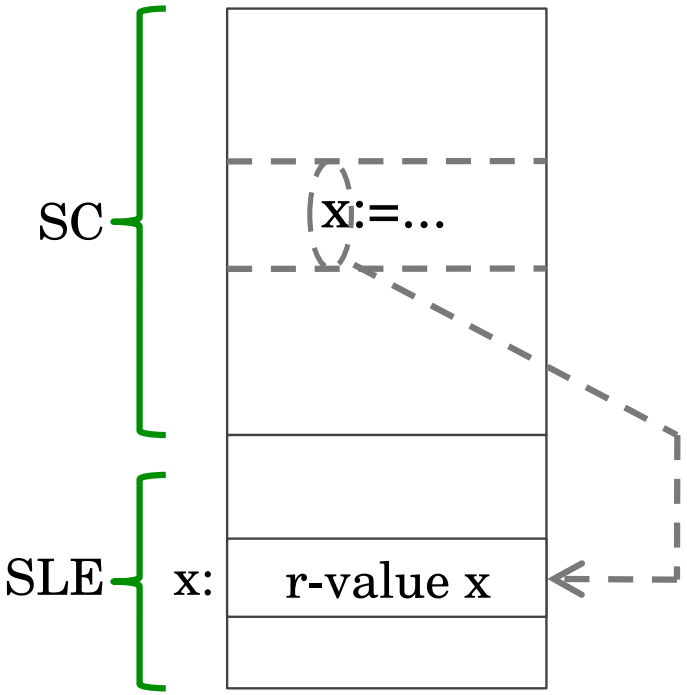
Static Local Environment

Example: static option in C creates static local environment

```
void f()
{
    static int x = 0;
    x++;
    printf("%d,", x);
    f();
}
...
while(1) { f(); } -> 1,2,3,4,5,
```

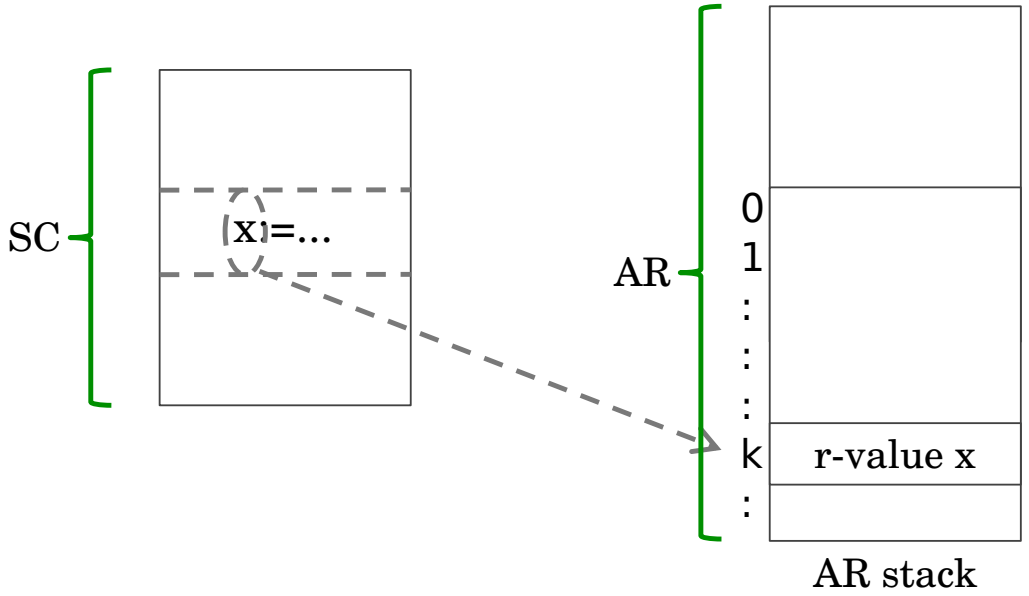
What happens without static?

Static Local environment - Implementation



- The table of static local environment: it's memorized only once and divided by all the calls of subprogram
- SLE is simply a sequence of r-value
- The names are offset inside the SLE

Dynamic Local environment - Implementation



- The local environment is a part of the activation record (AR): different calls of subprogram correspond to different instances of the local environment
- Again, the local names in the subprogram are compiled as offset, but this time inside the AR

Non local references

Example

```
procedure Q()
    begin
        ...
        x
        ...
    end
```

If x is not local, which binding is used for x?

Answer: rules of scoping

Dynamic scoping: rules of visibility are related to the execution (e.g., Lisp)

Static scoping: rules of visibility are related to the syntax of the program (e.g., C, Java, Pascal, ML...)

Outline

Static scoping - 1

Definition

Every identifier has a declaration that statically binds it. This binding is *constant at runtime*.

- The type of the identifier is known at compile time
- The location for the value of identifier can change at runtime (dynamic local environment) or not (static local environment)

Static scoping - 2

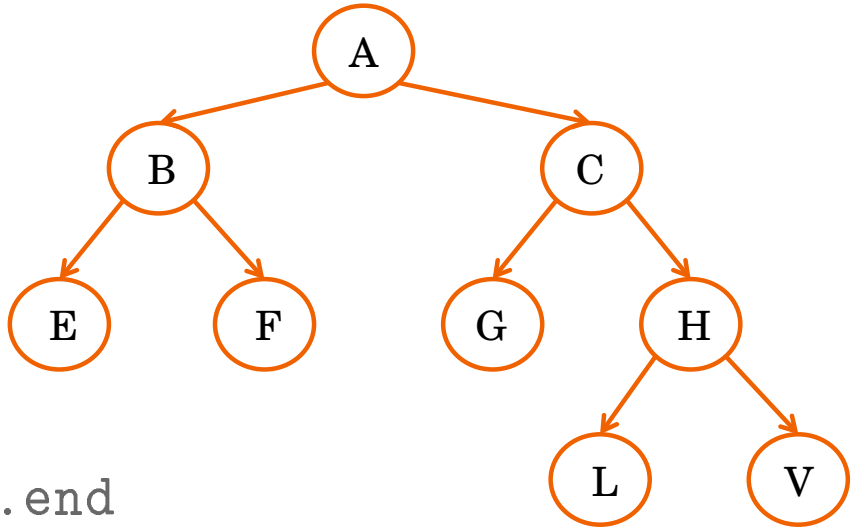
Scoping tree

For more rigorous analysis, for every program let's associate a tree called **scoping tree**:

- we give different names to blocks (the subprograms already have different names)
- nodes of the tree $\longrightarrow$ names of the blocks and subprograms
- `Q` is a child of `P` if
  - `Q` is a direct block of `P`
  - `Q` is a subprogram declared in `P`

Scoping tree example

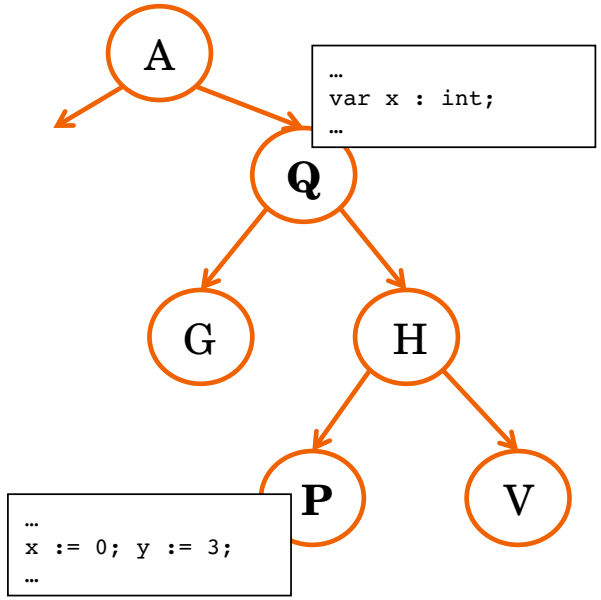
```
A: begin
    proc B;
        begin
            E: begin...end
            F: begin...end
        end {B}
    C: begin
        G: begin...end
        proc H;
            begin
                L: begin...end
                V: begin...end
            end {H}
    end {C}
end {A}
```



Rule of static scoping - 1

If x occurs in non local reference in the subprogram/block P

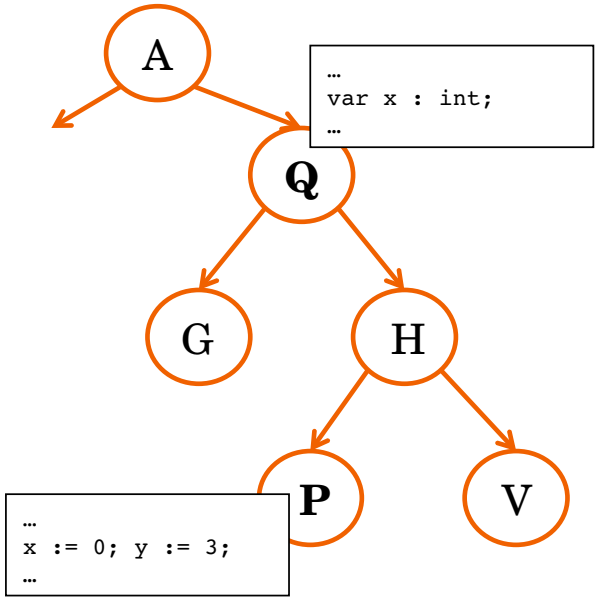
1. non local environment that provides a correct binding for x is the parent Q nearest to P in which x is declared
2. if there is no parent Q that declares x, an error is generated (remember, this control is made at compile time)



Rule of static scoping - 1

If `x` occurs in non local reference in the subprogram/block `P`

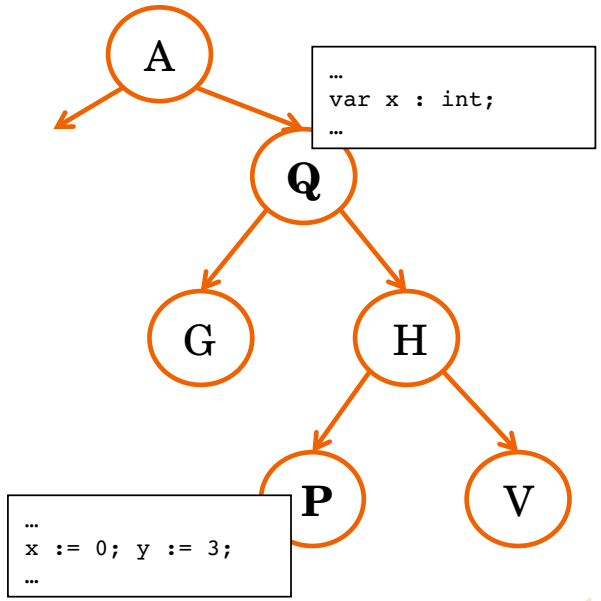
1. non local environment that provides a correct binding for `x` is the parent `Q` nearest to `P` in which `x` is declared
2. if there is no parent `Q` that declares `x`, an error is generated (remember, this control is made at compile time)



Rule of static scoping - 2

If the language defines a global environment outside of subprograms/blocks

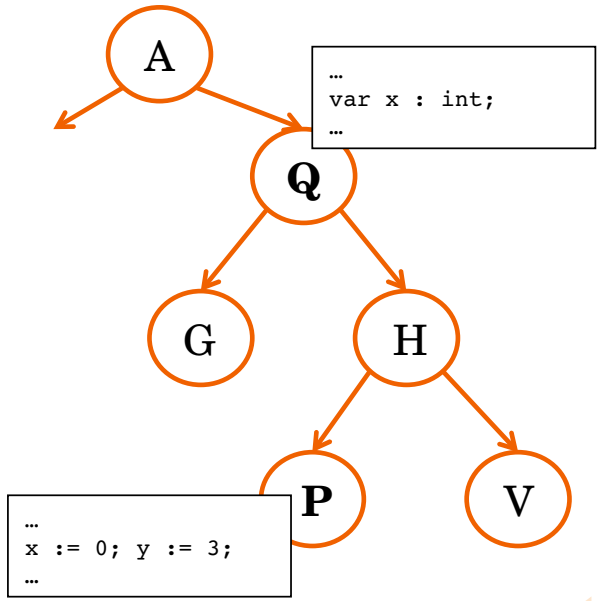
1. non local environment that provides a correct binding for `x` is the parent `Q` nearest to `P` in which `x` is declared (as before)
2. if there is no parent `Q` of `P` that declares `x`, then `x` is searched in the global environment
3. if not found an error is generated (at compile time)



Rule of static scoping - 2

If the language defines a global environment outside of subprograms/blocks

1. non local environment that provides a correct binding for `x` is the parent `Q` nearest to `P` in which `x` is declared (as before)
2. if there is no parent `Q` of `P` that declares `x`, then `x` is searched in the global environment
3. if not found an error is generated (at compile time)



Outline

Static scoping: semantics - 1

Environment 2.0

An environment a *sequence* of local environments:

$$\mathsf{Env} = \mathsf{List}(\mathsf{Ide} \longrightarrow \mathsf{DVal})$$
$$r = [r_0, r_1, \ldots, r_k]$$
$$\mathsf{DVal} = (\mathsf{Val} \cup \mathsf{Loc} \cup \mathsf{Com})$$

Rule of scoping

$r(x)$ is defined as follows:

- if $r_k(x)$ is defined, then $r_k(x)$, otherwise:
- if $r_{k-1}(x)$ is defined, then $r_{k-1}(x)$, otherwise:
- ...
- if $r_0(x)$ is defined, then $r_0(x)$, otherwise:
- ERROR

Static scoping: semantics - 1

Environment 2.0

An environment a *sequence* of local environments:

$$\text{Env} = \text{List}(\text{Ide} \longrightarrow \text{DVal})$$
$$r = [r_0, r_1, \ldots, r_k]$$
$$\text{DVal} = (\text{Val} \cup \text{Loc} \cup \text{Com})$$

Rule of scoping

$r(x)$ is defined as follows:

- if $r_k(x)$ is defined, then $r_k(x)$, otherwise:
- if $r_{k-1}(x)$ is defined, then $r_{k-1}(x)$, otherwise:
- ...
- if $r_0(x)$ is defined, then $r_0(x)$, otherwise:
- ERROR

Static scoping: semantics - 2

$$D\|\texttt{const } v = n\|_{[r_0,\ldots,r_k]s} = [r_0,\ldots,r_{k+1}]s \quad \text{where}$$

$$r_{k+1}(y) = \begin{cases} r_k(y) & \text{if } y \neq v \\ n, & \text{if } y = v \end{cases}$$

$$D\|\texttt{var } v := n\|_{[r_0,\ldots,r_k]s} = [r_0,\ldots,r_{k+1}]s' \quad \text{where}$$

$$r_{k+1}(y) = \begin{cases} r_k(y) & \text{if } y \neq v \\ l = \texttt{newmem s}, & \text{if } y = v \end{cases} \qquad s'(x) = \begin{cases} s(x) & \text{if } x \neq l \\ n, & \text{if } x = l \end{cases}$$

$$D\|\texttt{proc } P = C\|_{[r_0,\ldots,r_k]s} = [r_0,\ldots,r_{k+1}]s \quad \text{where}$$

$$r_{k+1}(y) = \begin{cases} r_k(y) & \text{if } y \neq P \\ C, & \text{if } y = P \end{cases}$$

Outline

Static scoping: implementation - 1

Problem

The Activation Record stack provides a temporal order between local environments (useless for static scoping), but gives no indication on the structure of the program.

Solution

- To each AR the *static chain pointer* (SCP) is added. The "static" information on the syntactic structure (scoping tree) is implemented through the SCP.
- A subprogram/block `Q` is parent of subprogram/block `P` in the scoping tree. Then, the SCP of an AR of `P` points to AR of `Q` according to the rule of static scoping.

Static scoping: implementation - 1

Problem

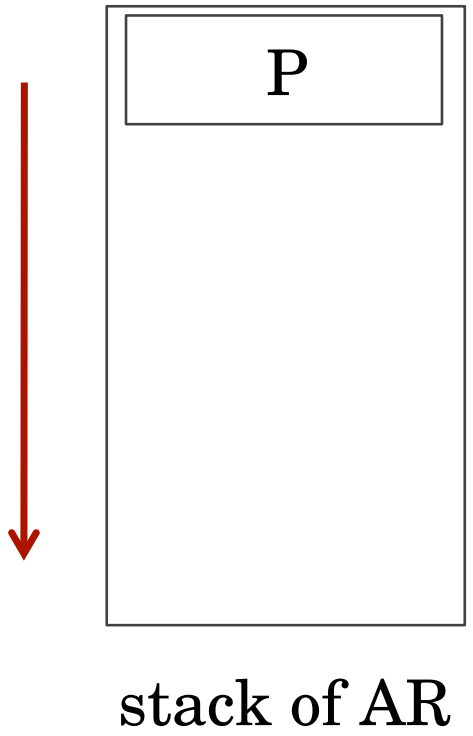
The Activation Record stack provides a temporal order between local environments (useless for static scoping), but gives no indication on the structure of the program.

Solution

- To each AR the *static chain pointer* (SCP) is added. The "static" information on the syntactic structure (scoping tree) is implemented through the SCP.
- A subprogram/block `Q` is parent of subprogram/block `P` in the scoping tree. Then, the SCP of an AR of `P` points to AR of `Q` according to the rule of static scoping.

Static scoping: implementation - 2

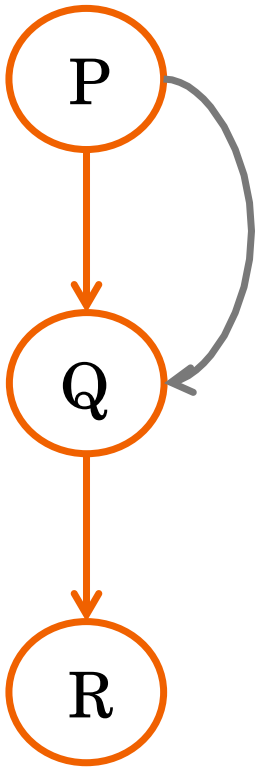
Suppose that Q $\Downarrow$ R; then, the AR of P is pushed in the stack of AR

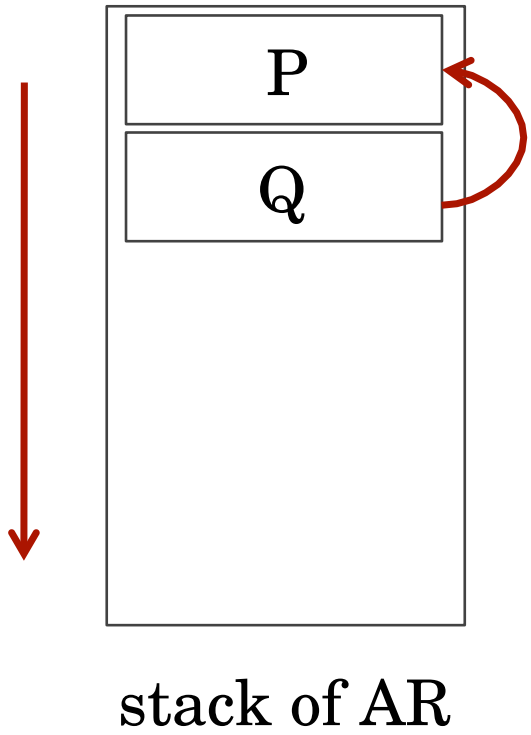


R is a child of Q but in the stack there are several occurrences of Q.

Static scoping: implementation - 2

Suppose that Q $\Downarrow$ R; then, the AR of P is pushed in the stack of AR

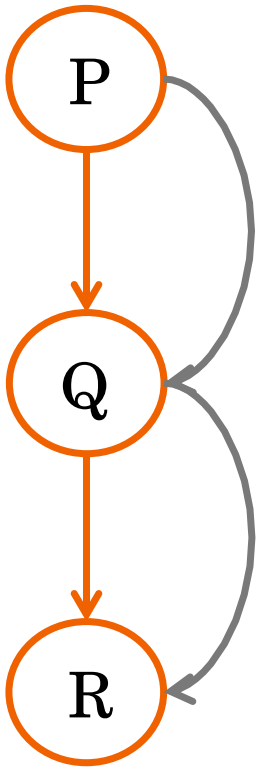


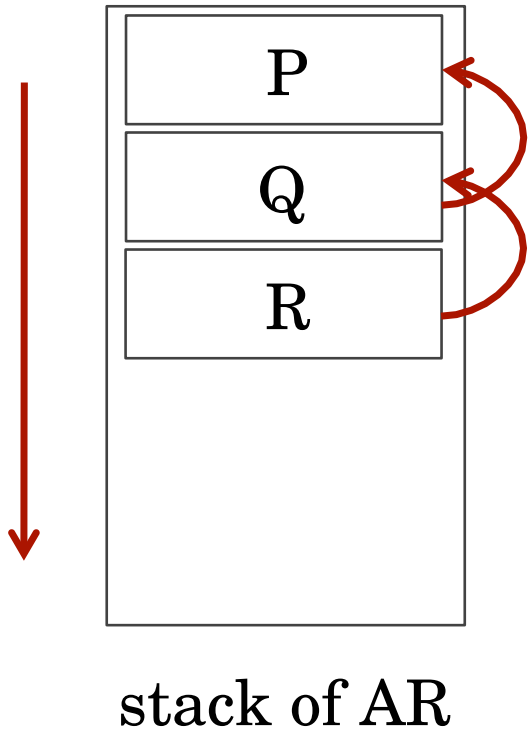


R is a child of Q but in the stack there are several occurrences of Q.

Static scoping: implementation - 2

Suppose that Q $\Downarrow$ R; then, the AR of P is pushed in the stack of AR

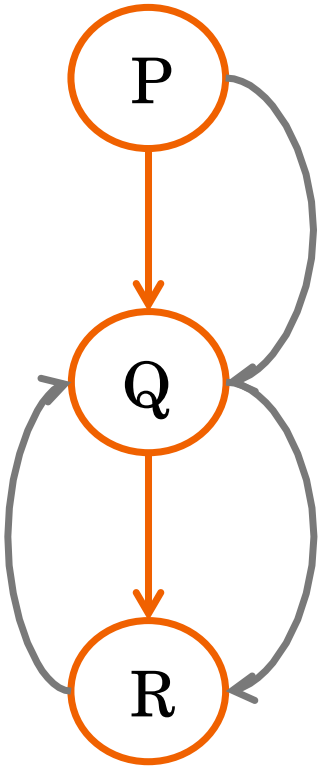


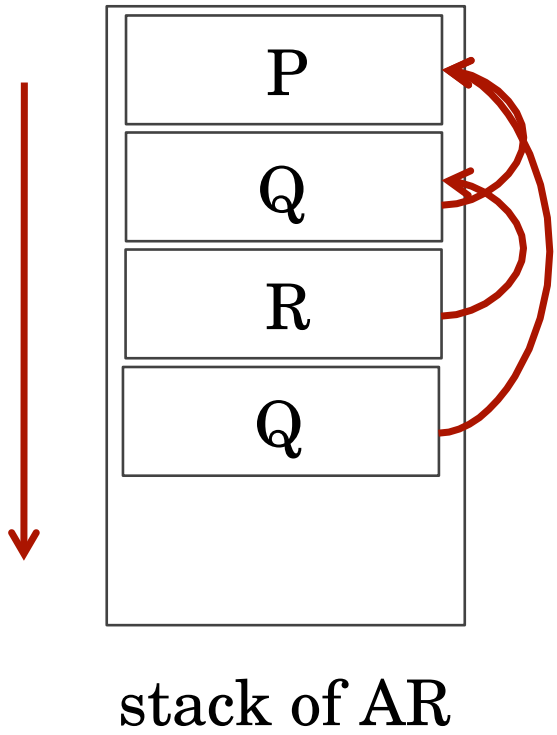


R is a child of Q but in the stack there are several occurrences of Q.

Static scoping: implementation - 2

Suppose that Q $\Downarrow$ R; then, the AR of P is pushed in the stack of AR

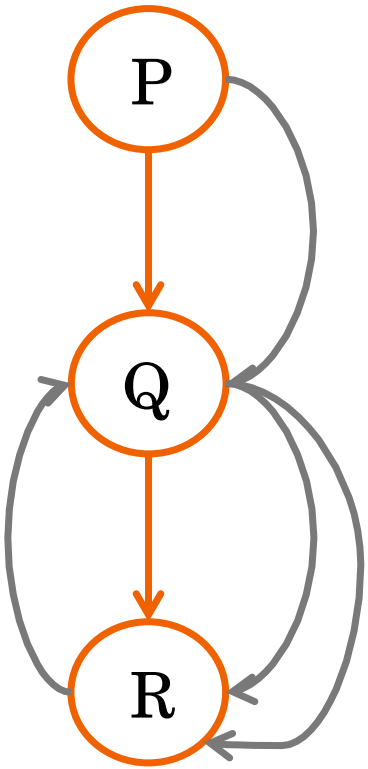


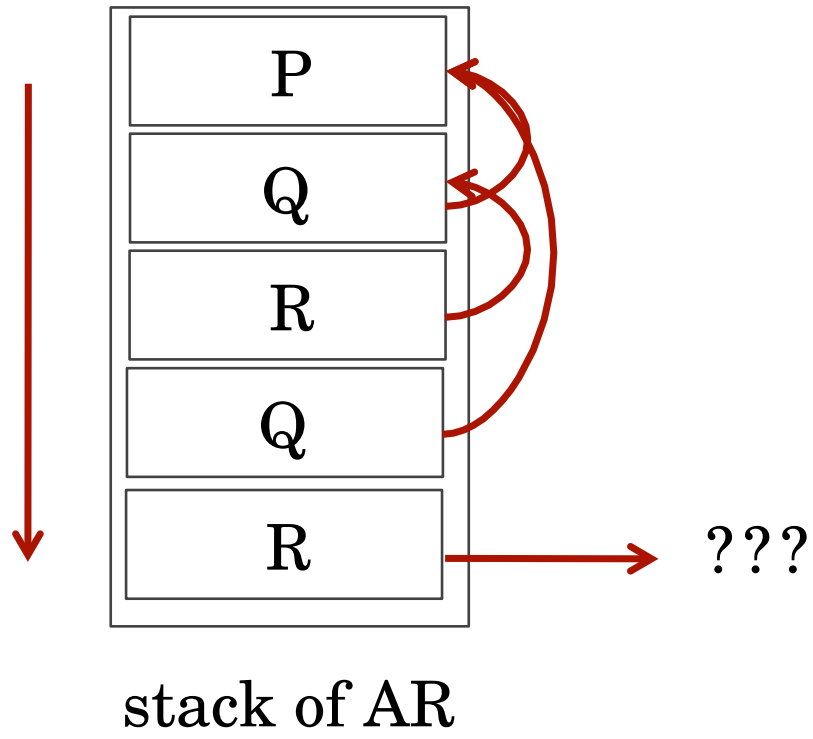


R is a child of Q but in the stack there are several occurrences of Q.

Static scoping: implementation - 2

Suppose that `Q` $\Downarrow$ `R`; the AR of P is pushed in the stack of AR



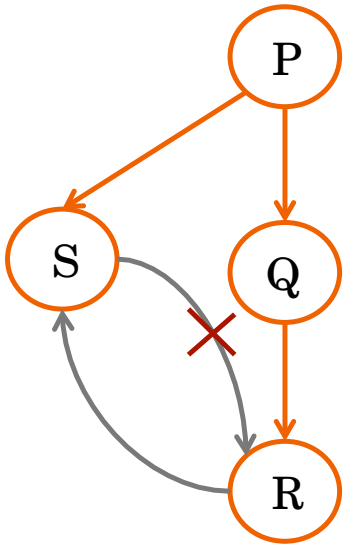


`R` is a child of `Q` but in the stack there are several occurrences of `Q`.

Outline

Algorithm to determine SCP

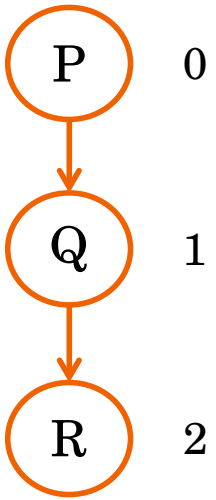
- Suppose α and β are nodes of the scoping tree, and suppose that $\alpha \Downarrow \beta$
- Then, the parent of β should be an ancestor of α (otherwise β would not be visible from α)




```
P: begin
     proc S; begin...end {S}
     proc Q;
          begin
               proc R; begin...end {R}
          end {Q}
end {P}
```

Algorithm to determine SCP

- Suppose α and β are nodes of the scoping tree, and suppose that $\alpha \Downarrow \beta$
- Then, the parent of β should be an ancestor of α (otherwise β would not be visible from α)
- Let's define $(\alpha, \beta) = \mathit{depth}(\alpha) - \mathit{depth}(\textbf{parent}(\beta))$



$\texttt{Q} \Downarrow \texttt{R}$ then $(\texttt{Q}, \texttt{R}) = 1 - 1 = 0$

$\texttt{R} \Downarrow \texttt{Q}$ then $(\texttt{R}, \texttt{Q}) = 2 - 0 = 2$

Algorithm to determine SCP

If $\texttt{P} \Downarrow \texttt{Q}$ then

1. The AR of Q ($AR_{\texttt{Q}}$) is put in the stack
2. The distance (P, Q) is calculated
3. The address a is reached by making (P, Q) steps starting from SCP of AR of the caller P; this is the address of an AR corresponding to a subprogram/block T that declares Q.
4. SCP of $AR_{\texttt{Q}}$ has value a

Algorithm to determine SCP

If $\texttt{P} \Downarrow \texttt{Q}$ then

1. The AR of Q ($AR_{\texttt{Q}}$) is put in the stack
2. The distance (P, Q) is calculated
3. The address a is reached by making (P, Q) steps starting from SCP of AR of the caller P; this is the address of an AR corresponding to a subprogram/block T that declares Q.
4. SCP of $AR_{\texttt{Q}}$ has value a

Algorithm to determine SCP

If P$\Downarrow$Q then

1. The AR of Q (AR_Q) is put in the stack
2. The distance (P, Q) is calculated
3. The address a is reached by making (P, Q) steps starting from SCP of AR of the caller P; this is the address of an AR corresponding to a subprogram/block T that declares Q.
4. SCP of AR_Q has value a

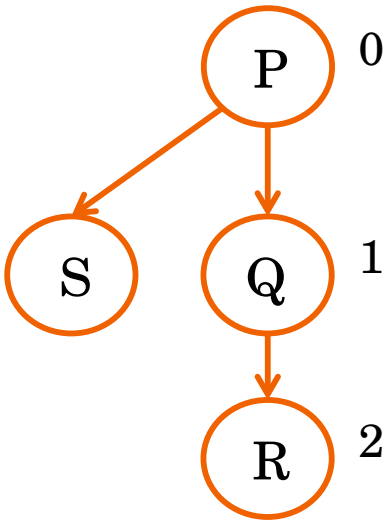
Algorithm to determine SCP

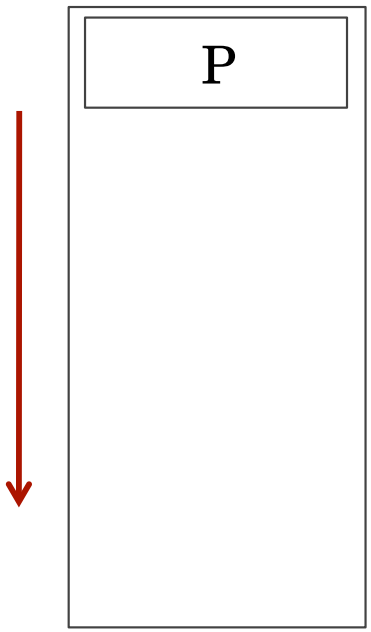
If P⇓Q then

1. The AR of Q (AR_Q) is put in the stack
2. The distance (P, Q) is calculated
3. The address a is reached by making (P, Q) steps starting from SCP of AR of the caller P; this is the address of an AR corresponding to a subprogram/block T that declares Q.
4. SCP of AR_Q has value a

Determining SCP: Example

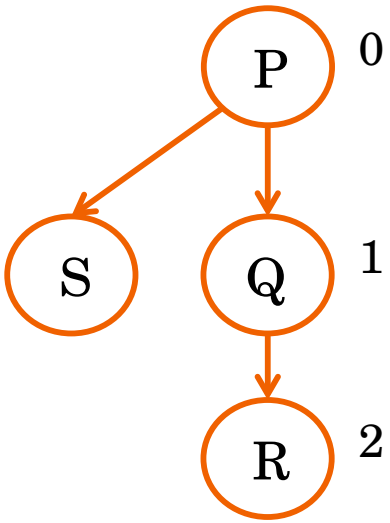
P



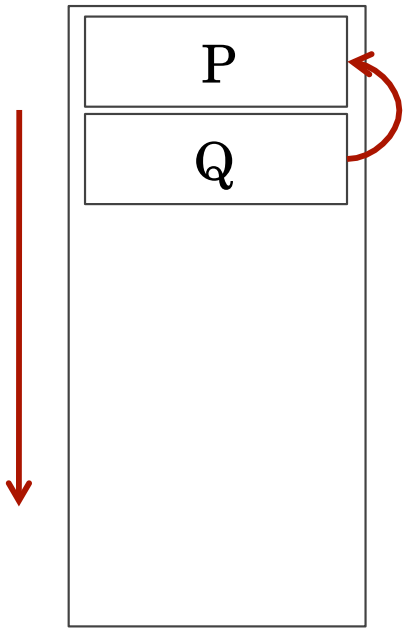


Determining SCP: Example

$$P \Downarrow Q$$

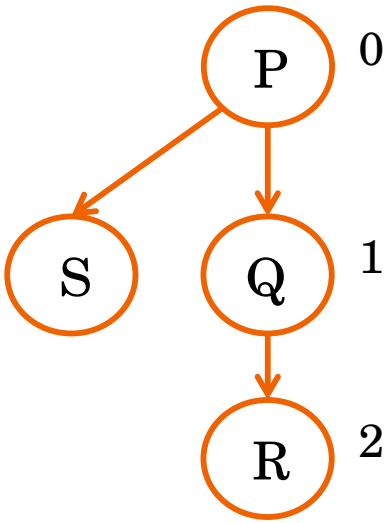


$$(P, Q) = 0$$



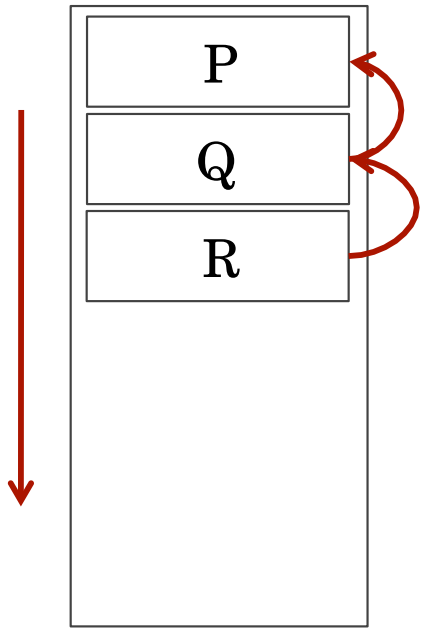
Determining SCP: Example

$$P \Downarrow Q \Downarrow R$$



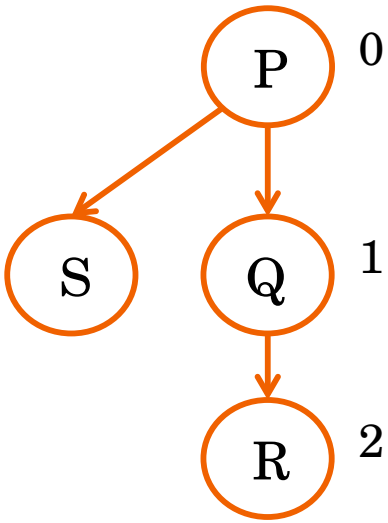
(P, Q) = 0

(Q, R) = 0



Determining SCP: Example

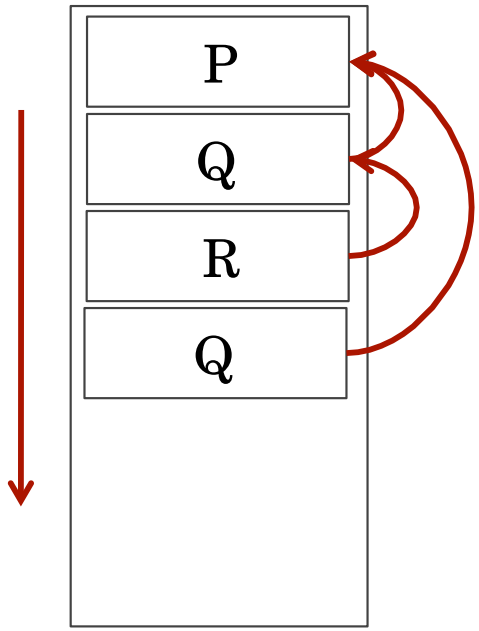
$$P \Downarrow Q \Downarrow R \Downarrow Q$$



(P, Q) = 0

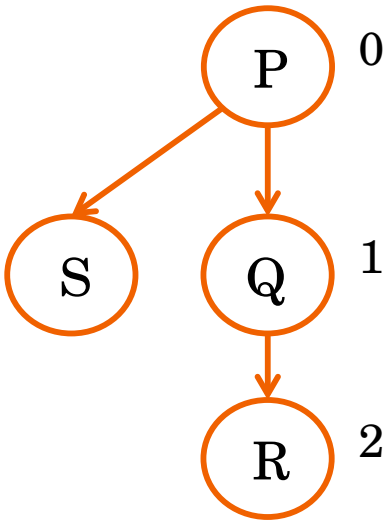
(Q, R) = 0

(R, Q) = 2



Determining SCP: Example

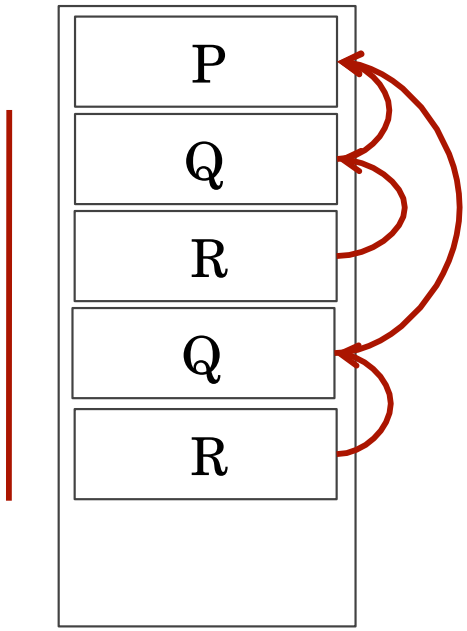
$$P \Downarrow Q \Downarrow R \Downarrow Q \Downarrow R$$



(P, Q) = 0

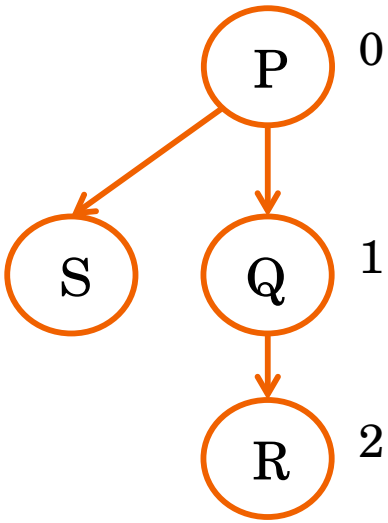
(Q, R) = 0

(R, Q) = 2



Determining SCP: Example

$$P \Downarrow Q \Downarrow R \Downarrow Q \Downarrow R \Downarrow S$$

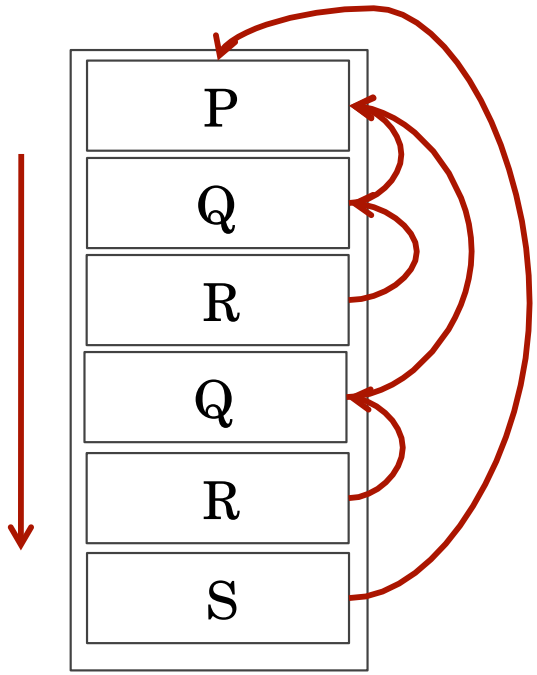


(P, Q) = 0

(Q, R) = 0

(R, Q) = 2

(R, S) = 2



Outline

Calling a subprogram: semantics

Given $r = [r_0, r_1, \ldots, r_k]$,

$$C[\![\texttt{call P}]\!]_{rs} = C[\![Cmd]\!]_{r's}$$

where

- program P is declared as $\texttt{proc P} = Cmd$
- $Cmd = r(\texttt{P}) \in Com$
- $r' = [r_0, r_1, \ldots, r_h, r_\varepsilon]$ where:
  - $h = depth(r, \texttt{P})$, or r_h is the "deepest" environment where P is defined,
  - P is defined in r_h,
  - P is not defined in $r_{h+1}, r_{h+2}, \ldots, r_k$
- r_ε is a new local (empty) environment for Cmd

Calling a subprogram: semantics

Given $r = [r_0, r_1, \ldots, r_k]$,

$$C[\![\texttt{call P}]\!]_{rs} = C[\![Cmd]\!]_{r's}$$

where

- program P is declared as $\texttt{proc P} = Cmd$
- $Cmd = r(\texttt{P}) \in Com$
- $r' = [r_0, r_1, \ldots, r_h, r_\varepsilon]$ where:
  - $h = depth(r, \texttt{P})$, or r_h is the "deepest" environment where P is defined,
  - P is defined in r_h,
  - P is not defined in $r_{h+1}, r_{h+2}, \ldots, r_k$
- r_ε is a new local (empty) environment for Cmd

Calling a subprogram: semantics

Given $r = [r_0, r_1, \ldots, r_k]$,

$$C[\![\texttt{call P}]\!]_{rs} = C[\![Cmd]\!]_{r's}$$

where

- program P is declared as $\texttt{proc P} = Cmd$
- $Cmd = r(\texttt{P}) \in Com$
- $r' = [r_0, r_1, \ldots, r_h, r_\varepsilon]$ where:
  - $h = depth(r, \texttt{P})$, or r_h is the "deepest" environment where P is defined,
    - P is defined in r_h,
    - P is not defined in $r_{h+1}, r_{h+2}, \ldots, r_k$
  - r_ε is a new local (empty) environment for Cmd

Calling a subprogram: semantics

Given $r = [r_0, r_1, \ldots, r_k]$,

$$C[\![\texttt{call P}]\!]_{rs} = C[\![Cmd]\!]_{r's}$$

where

- program P is declared as $\texttt{proc P} = Cmd$
- $Cmd = r(\texttt{P}) \in Com$
- $r' = [r_0, r_1, \ldots, r_h, r_\varepsilon]$ where:
  - $h = depth(r, \texttt{P})$, or r_h is the "deepest" environment where P is defined,
  - P is defined in r_h,
  - P is not defined in $r_{h+1}, r_{h+2}, \ldots, r_k$
- r_ε is a new local (empty) environment for Cmd

Calling a subprogram: semantics

Given $r = [r_0, r_1, \ldots, r_k]$,

$$C[\![\texttt{call P}]\!]_{rs} = C[\![Cmd]\!]_{r's}$$

where

- program `P` is declared as `proc P` $= Cmd$
- $Cmd = r(\texttt{P}) \in Com$
- $r' = [r_0, r_1, \ldots, r_h, r_\varepsilon]$ where:
  - $h = depth(r, \texttt{P})$, or r_h is the "deepest" environment where `P` is defined,
  - `P` is defined in r_h,
  - `P` is not defined in $r_{h+1}, r_{h+2}, \ldots, r_k$
- r_ε is a new local (empty) environment for Cmd

Calling a subprogram: semantics

Given $r = [r_0, r_1, \ldots, r_k]$,

$$C[\![\texttt{call P}]\!]_{rs} = C[\![Cmd]\!]_{r's}$$

where

- program P is declared as $\texttt{proc P} = Cmd$
- $Cmd = r(\texttt{P}) \in Com$
- $r' = [r_0, r_1, \ldots, r_h, r_\varepsilon]$ where:
  - $h = depth(r, \texttt{P})$, or r_h is the "deepest" environment where P is defined,
  - P is defined in r_h,
  - P is not defined in $r_{h+1}, r_{h+2}, \ldots, r_k$
- r_ε is a new local (empty) environment for Cmd

Non local references

- Suppose that a subprogram/block `P` is using a name `n`
- Define $(\texttt{P}, \texttt{n}) = \mathit{depth}(\texttt{P}) - \mathit{depth}(\text{subprg./blk that declares } \texttt{n})$

Non local references

Every non local reference `n` in the subprogram/block `P` is represented as

<x, y>

where `x` = (`P`, `n`) and `y` = position (offset) of `n` in the template of AR of the subprogram/block that declares `n`.
If `x` = 0, then `n` is local and is compiled simply as `y`.

Non local references

- Suppose that a subprogram/block `P` is using a name `n`
- Define $(\texttt{P}, \texttt{n}) = \mathit{depth}(\texttt{P}) - \mathit{depth}(\text{subprg./blk that declares } \texttt{n})$

Non local references

Every non local reference `n` in the subprogram/block `P` is represented as

<x, y>

where `x` = (`P`, `n`) and `y` = position (offset) of `n` in the template of AR of the subprogram/block that declares `n`.
If `x` = 0, then `n` is local and is compiled simply as `y`.

Non local references: implementation

Observation

Given a subprogram `P`,

- the length of the static chain when `P` is executing is statically fixed
- the non-local reference to a variable `n` is resolved always at the same point in the chain

For the reason of efficiency, the static chain is often implemented as a vector (we call it **display**)
The access to the identifier with the "coordinates" `<x,y>` is calculated as:

```
display[x] + y
```

Passing the parameters - 1

Let's assume:

- dynamic local environment
- static scoping

Notation

`proc P(x)`: `x` is a formal parameter

`call P(e)`: `e` is an actual parameter or an argument

Formal parameters are treated as local variables (they are allocated in the activation record).

Passing the parameters - 1

Let's assume:

- dynamic local environment
- static scoping

Notation

`proc P(x):` `x` is a formal parameter

`call P(e):` `e` is an actual parameter or an argument

Formal parameters are treated as local variables (they are allocated in the activation record).

Example

```
proc P(x)
    begin
        int y;
        ...
    end
```

The local variables are `x` and `y`.

Passing the parameters - 2

- Notation `call P(x` $\Leftarrow_\alpha$ `e)` means that `P` is declared as `proc P(x)` and `P` is invoked as `call P(e)`
- α is type of passing the parameters

  Value: `call P(x` $\Leftarrow_{\texttt{Val}}$ `e)`

  Value-result: `call P(x` $\Leftarrow_{\texttt{Val-res}}$ `e)`

  Result: `call P(x` $\Leftarrow_{\texttt{Res}}$ `e)`

  Reference: `call P(x` $\Leftarrow_{\texttt{Ref}}$ `e)`

  Constant: `call P(x` $\Leftarrow_{\texttt{Const}}$ `e)`

  Name: `call P(x` $\Leftarrow_{\texttt{Name}}$ `e)`
- Note: `x`, `y` are variables, `e` is an arithmetical expression

Passing the parameters - 2

- Notation `call P(x` $\Leftarrow_\alpha$ `e)` means that `P` is declared as `proc P(x)` and `P` is invoked as `call P(e)`
- α is type of passing the parameters

  Value: `call P(x` $\Leftarrow_{\texttt{Val}}$ `e)`

  Value-result: `call P(x` $\Leftarrow_{\texttt{Val-res}}$ `e)`

  Result: `call P(x` $\Leftarrow_{\texttt{Res}}$ `e)`

  Reference: `call P(x` $\Leftarrow_{\texttt{Ref}}$ `e)`

  Constant: `call P(x` $\Leftarrow_{\texttt{Const}}$ `e)`

  Name: `call P(x` $\Leftarrow_{\texttt{Name}}$ `e)`
- Note: `x`, `y` are variables, `e` is an arithmetical expression

Passing the parameters - 2

- Notation `call P(x` $\Leftarrow_{\alpha}$ `e)` means that `P` is declared as `proc P(x)` and `P` is invoked as `call P(e)`
- α is type of passing the parameters

  Value: `call P(x` $\Leftarrow_{\texttt{Val}}$ `e)`

  Value-result: `call P(x` $\Leftarrow_{\texttt{Val-res}}$ `e)`

  Result: `call P(x` $\Leftarrow_{\texttt{Res}}$ `e)`

  Reference: `call P(x` $\Leftarrow_{\texttt{Ref}}$ `e)`

  Constant: `call P(x` $\Leftarrow_{\texttt{Const}}$ `e)`

  Name: `call P(x` $\Leftarrow_{\texttt{Name}}$ `e)`
- Note: `x`, `y` are variables, `e` is an arithmetical expression

Passing the parameters - 2

- Notation `call P(x` $\Leftarrow_{\alpha}$ `e)` means that `P` is declared as `proc P(x)` and `P` is invoked as `call P(e)`
- α is type of passing the parameters

  Value: `call P(x` $\Leftarrow_{\texttt{Val}}$ `e)`

  Value-result: `call P(x` $\Leftarrow_{\texttt{Val-res}}$ `e)`

  Result: `call P(x` $\Leftarrow_{\texttt{Res}}$ `e)`

  Reference: `call P(x` $\Leftarrow_{\texttt{Ref}}$ `e)`

  Constant: `call P(x` $\Leftarrow_{\texttt{Const}}$ `e)`

  Name: `call P(x` $\Leftarrow_{\texttt{Name}}$ `e)`
- Note: `x`, `y` are variables, `e` is an arithmetical expression

Passing the parameters - 2

- Notation `call P(x` $\Leftarrow_{\alpha}$ `e)` means that `P` is declared as `proc P(x)` and `P` is invoked as `call P(e)`
- α is type of passing the parameters

  Value: `call P(x` $\Leftarrow_{\texttt{Val}}$ `e)`

  Value-result: `call P(x` $\Leftarrow_{\texttt{Val-res}}$ `e)`

  Result: `call P(x` $\Leftarrow_{\texttt{Res}}$ `e)`

  Reference: `call P(x` $\Leftarrow_{\texttt{Ref}}$ `e)`

  Constant: `call P(x` $\Leftarrow_{\texttt{Const}}$ `e)`

  Name: `call P(x` $\Leftarrow_{\texttt{Name}}$ `e)`
- Note: `x`, `y` are variables, `e` is an arithmetical expression

Passing the parameters - 2

- Notation `call P(x` $\Leftarrow_{\alpha}$ `e)` means that `P` is declared as `proc P(x)` and `P` is invoked as `call P(e)`
- α is type of passing the parameters

  Value: `call P(x` $\Leftarrow_{\texttt{Val}}$ `e)`

  Value-result: `call P(x` $\Leftarrow_{\texttt{Val-res}}$ `e)`

  Result: `call P(x` $\Leftarrow_{\texttt{Res}}$ `e)`

  Reference: `call P(x` $\Leftarrow_{\texttt{Ref}}$ `e)`

  Constant: `call P(x` $\Leftarrow_{\texttt{Const}}$ `e)`

  Name: `call P(x` $\Leftarrow_{\texttt{Name}}$ `e)`

- Note: `x`, `y` are variables, `e` is an arithmetical expression

Passing the parameters - 2

- Notation `call P(x` $\Leftarrow_{\alpha}$ `e)` means that `P` is declared as `proc P(x)` and `P` is invoked as `call P(e)`
- α is type of passing the parameters

  Value: `call P(x` $\Leftarrow_{\texttt{Val}}$ `e)`

  Value-result: `call P(x` $\Leftarrow_{\texttt{Val-res}}$ `e)`

  Result: `call P(x` $\Leftarrow_{\texttt{Res}}$ `e)`

  Reference: `call P(x` $\Leftarrow_{\texttt{Ref}}$ `e)`

  Constant: `call P(x` $\Leftarrow_{\texttt{Const}}$ `e)`

  Name: `call P(x` $\Leftarrow_{\texttt{Name}}$ `e)`
- Note: `x`, `y` are variables, `e` is an arithmetical expression

Passing the parameters - 2

- Notation `call P(x` $\Leftarrow_{\alpha}$ `e)` means that `P` is declared as `proc P(x)` and `P` is invoked as `call P(e)`
- α is type of passing the parameters

Value:	`call P(x` $\Leftarrow_{\texttt{Val}}$ `e)`
Value-result:	`call P(x` $\Leftarrow_{\texttt{Val-res}}$ `e)`
Result:	`call P(x` $\Leftarrow_{\texttt{Res}}$ `e)`
Reference:	`call P(x` $\Leftarrow_{\texttt{Ref}}$ `e)`
Constant:	`call P(x` $\Leftarrow_{\texttt{Const}}$ `e)`
Name:	`call P(x` $\Leftarrow_{\texttt{Name}}$ `e)`

- Note: `x`, `y` are variables, `e` is an arithmetical expression

Passing the parameters - 2

- Notation `call P(x` $\Leftarrow_{\alpha}$ `e)` means that `P` is declared as `proc P(x)` and `P` is invoked as `call P(e)`
- α is type of passing the parameters

  Value: `call P(x` $\Leftarrow_{\texttt{Val}}$ `e)`
  Value-result: `call P(x` $\Leftarrow_{\texttt{Val-res}}$ `e)`
  Result: `call P(x` $\Leftarrow_{\texttt{Res}}$ `e)`
  Reference: `call P(x` $\Leftarrow_{\texttt{Ref}}$ `e)`
  Constant: `call P(x` $\Leftarrow_{\texttt{Const}}$ `e)`
  Name: `call P(x` $\Leftarrow_{\texttt{Name}}$ `e)`
- Note: `x`, `y` are variables, `e` is an arithmetical expression

Passing by value

$$\texttt{call P(x} \Leftarrow_{\texttt{Val}} \texttt{e)}$$

- The expression e is evaluated in the environment of the caller
- In the AR of P the value e is assigned to the variable x

$$C\|\texttt{call P(x} \Leftarrow_{\texttt{Val}} \texttt{e)}\|_{rs} = C\|Cmd\|_{r's'}$$

- l = newmem s
- v = $E\|e\|_{rs}$
- $r' = [r_0, \ldots, r_{depth(r,\texttt{P})}, r_{\texttt{P}}]$ with $r_{\texttt{P}}(x) = l$
- $s' = \text{updatemem}(s, l, v)$
- $Cmd = r(\texttt{P})$

Note: x is local in P!
It is already implemented in crème CAraMeL.

Passing by value

$$\texttt{call P(x} \Leftarrow_{\texttt{Val}} \texttt{e)}$$

- The expression e is evaluated in the environment of the caller
- In the AR of P the value e is assigned to the variable x

$$C\|\texttt{call P(x} \Leftarrow_{\texttt{Val}} \texttt{e)}\|_{rs} = C\|Cmd\|_{r's'}$$

- l = newmem s
- v = $E\|e\|_{rs}$
- $r' = [r_0, \ldots, r_{depth(r,\texttt{P})}, r_{\texttt{P}}]$ with $r_{\texttt{P}}(x) = l$
- $s' = \text{updatemem}(s, l, v)$
- $Cmd = r(\texttt{P})$

Note: x is local in P!
It is already implemented in crème CAraMeL.

Passing by value

$$\texttt{call P(x} \Leftarrow_{\texttt{Val}} \texttt{e)}$$

- The expression e is evaluated in the environment of the caller
- In the AR of P the value e is assigned to the variable x

$$C\|\texttt{call P(x} \Leftarrow_{\texttt{Val}} \texttt{e)}\|_{rs} = C\|Cmd\|_{r's'}$$

- $\texttt{l} = \texttt{newmem s}$
- $\texttt{v} = E\|e\|_{rs}$
- $r' = [r_0, \ldots, r_{depth(r,\texttt{P})}, r_{\texttt{P}}]$ with $r_{\texttt{P}}(x) = l$
- $s' = \text{updatemem}(s, l, v)$
- $Cmd = r(\texttt{P})$

Note: x is local in P!
It is already implemented in crème CAraMeL.

Passing by value

$$\texttt{call P(x} \Leftarrow_{\texttt{Val}} \texttt{e)}$$

- The expression e is evaluated in the environment of the caller
- In the AR of P the value e is assigned to the variable x

$$C\|\texttt{call P(x} \Leftarrow_{\texttt{Val}} \texttt{e)}\|_{rs} = C\|Cmd\|_{r's'}$$

- l = newmem s
- v = $E\|e\|_{rs}$
- $r' = [r_0, \ldots, r_{depth(r,\texttt{P})}, r_{\texttt{P}}]$ with $r_{\texttt{P}}(x) = l$
- $s' = \text{updatemem}(s, l, v)$
- $Cmd = r(\texttt{P})$

Note: x is local in P!
It is already implemented in crème CAraMeL.

Passing by value

$$\texttt{call P(x} \Leftarrow_{\texttt{Val}} \texttt{e)}$$

- The expression e is evaluated in the environment of the caller
- In the AR of P the value e is assigned to the variable x

$$C\|\texttt{call P(x} \Leftarrow_{\texttt{Val}} \texttt{e)}\|_{rs} = C\|Cmd\|_{r's'}$$

- $\texttt{l} = \texttt{newmem s}$
- $\texttt{v} = E\|e\|_{rs}$
- $r' = [r_0, \ldots, r_{depth(r,\texttt{P})}, r_{\texttt{P}}]$ with $r_{\texttt{P}}(x) = l$
- $s' = \text{updatemem}(s, l, v)$
- $Cmd = r(\texttt{P})$

Note: x is local in P!
It is already implemented in crème CAraMeL.

Passing by value

$$\texttt{call P(x} \Leftarrow_{\texttt{Val}} \texttt{e)}$$

- The expression e is evaluated in the environment of the caller
- In the AR of P the value e is assigned to the variable x

$$C\|\texttt{call P(x} \Leftarrow_{\texttt{Val}} \texttt{e)}\|_{rs} = C\|Cmd\|_{r's'}$$

- l = newmem s
- v = $E\|e\|_{rs}$
- $r' = [r_0, \ldots, r_{depth(r,\texttt{P})}, r_{\texttt{P}}]$ with $r_{\texttt{P}}(x) = l$
- $s' = \text{updatemem}(s, l, v)$
- $Cmd = r(\texttt{P})$

Note: x is local in P!

It is already implemented in crème CAraMeL.

Passing by value-result

$$\texttt{call P(x} \Leftarrow_{\texttt{Val-res}} \texttt{y)}$$

- The value of y is evaluated in the environment of the caller
- This value is assigned to the local variable x in P
- When P terminates, the value of x is copied to the variable y of the caller

$$C\|\texttt{call P(x} \Leftarrow_{\text{Val-res}} \texttt{y)}\|_{rs} = s'''$$

- $\texttt{v} = E\|y\|_{rs}$
- $\texttt{l} = \texttt{newmem s}$
- $r' = [r_0, \ldots, r_{depth(r,\texttt{P})}, r_{\texttt{P}}]$ with $r_{\texttt{P}}(x) = l$
- $s' = \text{updatemem}(s, l, v)$
- $Cmd = r(\texttt{P})$
- $C\|Cmd\|_{r's'} = s''$
- $s''' = \text{update}(s'', \Lambda\|y\|_{rs}, s''(l))$

Passing by value-result

$$\texttt{call P(x} \Leftarrow_{\texttt{Val-res}} \texttt{y)}$$

- The value of `y` is evaluated in the environment of the caller
- This value is assigned to the local variable `x` in `P`
- When `P` terminates, the value of `x` is copied to the variable `y` of the caller

$$C\|\texttt{call P(x} \Leftarrow_{\text{Val-res}} \texttt{y)}\|_{rs} = s'''$$

- $\texttt{v} = E\|y\|_{rs}$
- $\texttt{l} = \texttt{newmem s}$
- $r' = [r_0, \ldots, r_{depth(r,\texttt{P})}, r_{\texttt{P}}]$ with $r_{\texttt{P}}(x) = l$
- $s' = \text{updatemem}(s, l, v)$
- $Cmd = r(\texttt{P})$
- $C\|Cmd\|_{r's'} = s''$
- $s''' = \text{update}(s'', \Lambda\|y\|_{rs}, s''(l))$

Passing by value-result

$$\texttt{call P(x} \Leftarrow_{\texttt{Val-res}} \texttt{y)}$$

- The value of `y` is evaluated in the environment of the caller
- This value is assigned to the local variable `x` in `P`
- When `P` terminates, the value of `x` is copied to the variable `y` of the caller

$$C\|\texttt{call P(x} \Leftarrow_{\text{Val-res}} \texttt{y)}\|_{rs} = s'''$$

- $\texttt{v} = E\|y\|_{rs}$
- $\texttt{l} = \texttt{newmem s}$
- $r' = [r_0, \ldots, r_{depth(r,\texttt{P})}, r_{\texttt{P}}]$ with $r_{\texttt{P}}(x) = l$
- $s' = \text{updatemem}(s, l, v)$
- $Cmd = r(\texttt{P})$
- $C\|Cmd\|_{r's'} = s''$
- $s''' = \text{update}(s'', \Lambda\|y\|_{rs}, s''(l))$

Passing by value-result

$$\texttt{call P(x} \Leftarrow_{\texttt{Val-res}} \texttt{y)}$$

- The value of `y` is evaluated in the environment of the caller
- This value is assigned to the local variable `x` in `P`
- When `P` terminates, the value of `x` is copied to the variable `y` of the caller

$$C\|\texttt{call P(x} \Leftarrow_{\text{Val-res}} \texttt{y)}\|_{rs} = s'''$$

- $\texttt{v} = E\|y\|_{rs}$
- $\texttt{l} = \texttt{newmem s}$
- $r' = [r_0, \ldots, r_{depth(r,\texttt{P})}, r_{\texttt{P}}]$ with $r_{\texttt{P}}(x) = l$
- $s' = \text{updatemem}(s, l, v)$
- $Cmd = r(\texttt{P})$
- $C\|Cmd\|_{r's'} = s''$
- $s''' = \text{update}(s'', \Lambda\|y\|_{rs}, s''(l))$

Passing by value-result

$$\texttt{call P(x} \Leftarrow_{\texttt{Val-res}} \texttt{y)}$$

- The value of `y` is evaluated in the environment of the caller
- This value is assigned to the local variable `x` in `P`
- When `P` terminates, the value of `x` is copied to the variable `y` of the caller

$$C\|\texttt{call P(x} \Leftarrow_{\text{Val-res}} \texttt{y)}\|_{rs} = s'''$$

- $\texttt{v} = E\|y\|_{rs}$
- $\texttt{l} = \texttt{newmem s}$
- $r' = [r_0, \ldots, r_{depth(r,\texttt{P})}, r_{\texttt{P}}]$ with $r_{\texttt{P}}(x) = l$
- $s' = \text{updatemem}(s, l, v)$
- $Cmd = r(\texttt{P})$
- $C\|Cmd\|_{r's'} = s''$
- $s''' = \text{update}(s'', \Lambda\|y\|_{rs}, s''(l))$

Passing by value-result

$$\texttt{call P(x} \Leftarrow_{\texttt{Val-res}} \texttt{y)}$$

- The value of `y` is evaluated in the environment of the caller
- This value is assigned to the local variable `x` in `P`
- When `P` terminates, the value of `x` is copied to the variable `y` of the caller

$$C\|\texttt{call P(x} \Leftarrow_{\text{Val-res}} \texttt{y)}\|_{rs} = s'''$$

- $\texttt{v} = E\|y\|_{rs}$
- $\texttt{l} = \texttt{newmem s}$
- $r' = [r_0, \ldots, r_{depth(r,\texttt{P})}, r_{\texttt{P}}]$ with $r_{\texttt{P}}(x) = l$
- $s' = \text{updatemem}(s, l, v)$
- $Cmd = r(\texttt{P})$
- $C\|Cmd\|_{r's'} = s''$
- $s''' = \text{update}(s'', \Lambda\|y\|_{rs}, s''(l))$

Passing by value-result

$$\texttt{call P(x} \Leftarrow_{\texttt{Val-res}} \texttt{y)}$$

- The value of `y` is evaluated in the environment of the caller
- This value is assigned to the local variable `x` in `P`
- When `P` terminates, the value of `x` is copied to the variable `y` of the caller

$$C\|\texttt{call P(x} \Leftarrow_{\text{Val-res}} \texttt{y)}\|_{rs} = s'''$$

- $\texttt{v} = E\|y\|_{rs}$
- $\texttt{l} = \texttt{newmem s}$
- $r' = [r_0, \ldots, r_{depth(r,\texttt{P})}, r_{\texttt{P}}]$ with $r_{\texttt{P}}(x) = l$
- $s' = \text{updatemem}(s, l, v)$
- $Cmd = r(\texttt{P})$
- $C\|Cmd\|_{r's'} = s''$
- $s''' = \text{update}(s'', \Lambda\|y\|_{rs}, s''(l))$

Passing by value-result

$$\texttt{call P(x} \Leftarrow_{\texttt{Val-res}} \texttt{y)}$$

- The value of `y` is evaluated in the environment of the caller
- This value is assigned to the local variable `x` in `P`
- When `P` terminates, the value of `x` is copied to the variable `y` of the caller

$$C\|\texttt{call P(x} \Leftarrow_{\text{Val-res}} \texttt{y)}\|_{rs} = s'''$$

- $\texttt{v} = E\|y\|_{rs}$
- $\texttt{l} = \texttt{newmem s}$
- $r' = [r_0, \ldots, r_{depth(r,\texttt{P})}, r_{\texttt{P}}]$ with $r_{\texttt{P}}(x) = l$
- $s' = \text{updatemem}(s, l, v)$
- $Cmd = r(\texttt{P})$
- $C\|Cmd\|_{r's'} = s''$
- $s''' = \text{update}(s'', \Lambda\|y\|_{rs}, s''(l))$

Passing by value-result

$$\texttt{call P(x} \Leftarrow_{\texttt{Val-res}} \texttt{y)}$$

- The value of `y` is evaluated in the environment of the caller
- This value is assigned to the local variable `x` in `P`
- When `P` terminates, the value of `x` is copied to the variable `y` of the caller

$$C\|\texttt{call P(x} \Leftarrow_{\text{Val-res}} \texttt{y)}\|_{rs} = s'''$$

- $\texttt{v} = E\|y\|_{rs}$
- $\texttt{l} = \texttt{newmem s}$
- $r' = [r_0, \ldots, r_{depth(r,\texttt{P})}, r_{\texttt{P}}]$ with $r_{\texttt{P}}(x) = l$
- $s' = \text{updatemem}(s, l, v)$
- $Cmd = r(\texttt{P})$
- $C\|Cmd\|_{r's'} = s''$
- $s''' = \text{update}(s'', \Lambda\|y\|_{rs}, s''(l))$

Passing by value-result

$$\texttt{call P(x} \Leftarrow_{\texttt{Val-res}} \texttt{y)}$$

- The value of `y` is evaluated in the environment of the caller
- This value is assigned to the local variable `x` in `P`
- When `P` terminates, the value of `x` is copied to the variable `y` of the caller

$$C\|\texttt{call P(x} \Leftarrow_{\text{Val-res}} \texttt{y)}\|_{rs} = s'''$$

- $\texttt{v} = E\|y\|_{rs}$
- $\texttt{l} = \texttt{newmem s}$
- $r' = [r_0, \ldots, r_{depth(r,\texttt{P})}, r_{\texttt{P}}]$ with $r_{\texttt{P}}(x) = l$
- $s' = \text{updatemem}(s, l, v)$
- $Cmd = r(\texttt{P})$
- $C\|Cmd\|_{r's'} = s''$
- $s''' = \text{update}(s'', \Lambda\|y\|_{rs}, s''(l))$

Passing by result

$$\texttt{call P(x} \Leftarrow_{\texttt{Res}} \texttt{y)}$$

- When P terminates, x is copied to the variable y
- Initial value of x is not specified
- The semantics is like in passing by value-result without the evaluation of y

Passing by result

$$\texttt{call P(x} \Leftarrow_{\texttt{Res}} \texttt{y)}$$

- When `P` terminates, `x` is copied to the variable `y`
- Initial value of `x` is not specified
- The semantics is like in passing by value-result without the evaluation of `y`

Passing by result

$$\texttt{call P(x} \Leftarrow_{\texttt{Res}} \texttt{y)}$$

- When `P` terminates, `x` is copied to the variable `y`
- Initial value of `x` is not specified
- The semantics is like in passing by value-result without the evaluation of `y`

Passing by reference

$$\texttt{call P(x} \Leftarrow_{\texttt{Ref}} \texttt{y)}$$

- The location `l` of `y` is evaluated in the environment of the caller
- The location of `x` in `P` is set to `l`

$$C\|\texttt{call P(x} \Leftarrow_{\texttt{Ref}} \texttt{y)}\|_{rs} = C\|Cmd\|_{r's}$$

- $\texttt{l} = \Lambda\|y\|_{rs}$
- $r' = [r_0, \ldots, r_{depth(r,\texttt{P})}, r_{\texttt{P}}]$ with $r_{\texttt{P}}(x) = l$
- $Cmd = r(\texttt{P})$

Passing by reference

$$\texttt{call P(x} \Leftarrow_{\texttt{Ref}} \texttt{y)}$$

- The location `l` of `y` is evaluated in the environment of the caller
- The location of `x` in `P` is set to `l`

$$C\|\texttt{call P(x} \Leftarrow_{\texttt{Ref}} \texttt{y)}\|_{rs} = C\|Cmd\|_{r's}$$

- $\texttt{l} = \Lambda\|y\|_{rs}$
- $r' = [r_0, \ldots, r_{depth(r,\texttt{P})}, r_{\texttt{P}}]$ with $r_{\texttt{P}}(x) = l$
- $Cmd = r(\texttt{P})$

Passing by reference

$$\texttt{call P(x} \Leftarrow_{\texttt{Ref}} \texttt{y)}$$

- The location `l` of `y` is evaluated in the environment of the caller
- The location of `x` in `P` is set to `l`

$$C\|\texttt{call P(x} \Leftarrow_{\texttt{Ref}} \texttt{y)}\|_{rs} = C\|Cmd\|_{r's}$$

- $\texttt{l} = \Lambda\|y\|_{rs}$
- $r' = [r_0, \ldots, r_{depth(r,\texttt{P})}, r_{\texttt{P}}]$ with $r_{\texttt{P}}(x) = l$
- $Cmd = r(\texttt{P})$

Passing by reference

$$\texttt{call P(x} \Leftarrow_{\texttt{Ref}} \texttt{y)}$$

- The location `l` of `y` is evaluated in the environment of the caller
- The location of `x` in `P` is set to `l`

$$C\|\texttt{call P(x} \Leftarrow_{\texttt{Ref}} \texttt{y)}\|_{rs} = C\|Cmd\|_{r's}$$

- $\texttt{l} = \Lambda\|y\|_{rs}$
- $r' = [r_0, \ldots, r_{depth(r,\texttt{P})}, r_{\texttt{P}}]$ with $r_{\texttt{P}}(x) = l$
- $Cmd = r(\texttt{P})$

Passing by constant

$$\texttt{call P(x} \Leftarrow_{\texttt{Const}} \texttt{e)}$$

- The value of e is evaluated in the environment of the caller
- This value is assigned to the local variable x in P
- Inside P, values cannot be assigned to x
- It can be implemented in a similar way to the passing by reference

Passing by constant

$$\texttt{call P(x} \Leftarrow_{\texttt{Const}} \texttt{e)}$$

- The value of e is evaluated in the environment of the caller
- This value is assigned to the local variable x in P
- Inside P, values cannot be assigned to x
- It can be implemented in a similar way to the passing by reference

Passing by constant

$$\texttt{call P(x} \Leftarrow_{\texttt{Const}} \texttt{e)}$$

- The value of e is evaluated in the environment of the caller
- This value is assigned to the local variable x in P
- Inside P, values cannot be assigned to x
- It can be implemented in a similar way to the passing by reference

Passing by constant

$$\texttt{call P(x} \Leftarrow_{\texttt{Const}} \texttt{e)}$$

- The value of e is evaluated in the environment of the caller
- This value is assigned to the local variable x in P
- Inside P, values cannot be assigned to x
- It can be implemented in a similar way to the passing by reference

Passing by name

$$\texttt{call P(x} \Leftarrow_{\texttt{Name}} \texttt{y)}$$

- Create a new couple <e, r>, where r is an environment of the caller
- Every time x should be evaluated, e is getting evaluated instead, in the environment r, and put instead of x
- Inside P, values cannot be assigned to x

Passing by name

$$\texttt{call P(x} \Leftarrow_{\texttt{Name}} \texttt{y)}$$

- Create a new couple <e, r>, where r is an environment of the caller
- Every time x should be evaluated, e is getting evaluated instead, in the environment r, and put instead of x
- Inside P, values cannot be assigned to x

Passing by name

$$\texttt{call P(x} \Leftarrow_{\texttt{Name}} \texttt{y)}$$

- Create a new couple <e, *r*>, where *r* is an environment of the caller
- Every time x should be evaluated, e is getting evaluated instead, in the environment *r*, and put instead of x
- Inside P, values cannot be assigned to x