

Declarative Computation Model

Kernel language semantics

(Non-)Suspendable statements (VRH 2.4.3-2.4.4)

Carlos Varela

RPI

March 8, 2012

Adapted with permission from:

Seif Haridi

KTH

Peter Van Roy

UCL

Sequential declarative computation model

- The kernel language semantics
 - The environment: maps textual variable names (variable identifiers) into entities in the store
 - Abstract machine consists of an execution stack of semantic statements transforming the store
 - Interpretation (execution) of the kernel language elements (statements) by the use of an abstract machine
 - Non-suspendable statements
 - Suspendable statements

Kernel language syntax

The following defines the syntax of a statement, $\langle s \rangle$ denotes a statement

$\langle s \rangle ::=$	<code>skip</code>	<i>empty statement</i>
	<code>$\langle x \rangle = \langle y \rangle$</code>	<i>variable-variable binding</i>
	<code>$\langle x \rangle = \langle v \rangle$</code>	<i>variable-value binding</i>
	<code>$\langle s_1 \rangle \langle s_2 \rangle$</code>	<i>sequential composition</i>
	<code>local $\langle x \rangle$ in $\langle s_1 \rangle$ end</code>	<i>declaration</i>
	<code>if $\langle x \rangle$ then $\langle s_1 \rangle$ else $\langle s_2 \rangle$ end</code>	<i>conditional</i>
	<code>{ $\langle x \rangle \langle y_1 \rangle \dots \langle y_n \rangle$ }</code>	<i>procedural application</i>
	<code>case $\langle x \rangle$ of $\langle \text{pattern} \rangle$ then $\langle s_1 \rangle$ else $\langle s_2 \rangle$ end</code>	<i>pattern matching</i>
$\langle v \rangle ::=$	<code>proc { \$ $\langle y_1 \rangle \dots \langle y_n \rangle$ } $\langle s_1 \rangle$ end ...</code>	<i>value expression</i>
$\langle \text{pattern} \rangle ::=$...	

Computations (abstract machine)

- A computation defines how the execution state is transformed step by step from the initial state to the final state
- A *single assignment store* σ is a set of store variables, a variable may be unbound, bound to a partial value, or bound to a group of other variables
- An *environment* E is mapping from variable identifiers to variables or values in σ , e.g. $\{X \rightarrow x_1, Y \rightarrow x_2\}$
- A *semantic statement* is a pair
($\langle s \rangle, E$) where $\langle s \rangle$ is a statement
- ST is a stack of semantic statements

Computations (abstract machine)

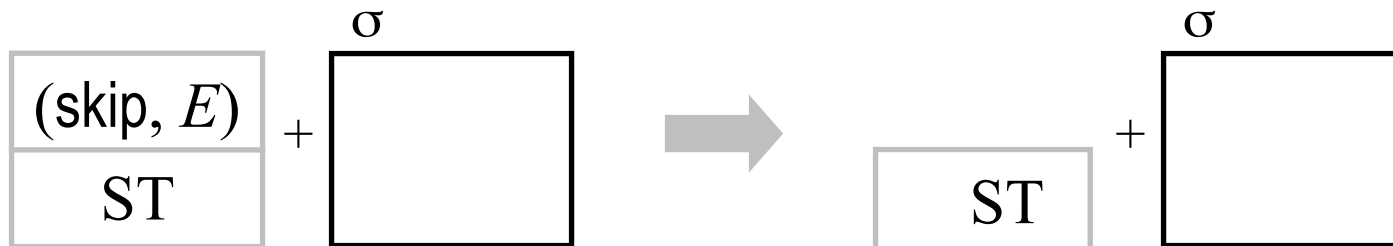
- A computation defines how the execution state is transformed step by step from the initial state to the final state
- The *execution state* is a pair
 (ST , σ)
 - where ST is a *stack of semantic statements* and σ is a *single assignment store*
- A *computation* is a sequence of execution states
 $(ST_0 , \sigma_0) \rightarrow (ST_1 , \sigma_1) \rightarrow (ST_2 , \sigma_2) \rightarrow \dots$

Semantics

- To execute a program (i.e., a statement) $\langle s \rangle$ the initial execution state is
 $([\langle s \rangle, \emptyset] , \emptyset)$
- ST has a single semantic statement $(\langle s \rangle, \emptyset)$
- The environment E is empty, and the store σ is empty
- $[\dots]$ denotes the stack
- At each step the first element of ST is popped and execution proceeds according to the form of the element
- The final execution state (if any) is a state in which ST is empty

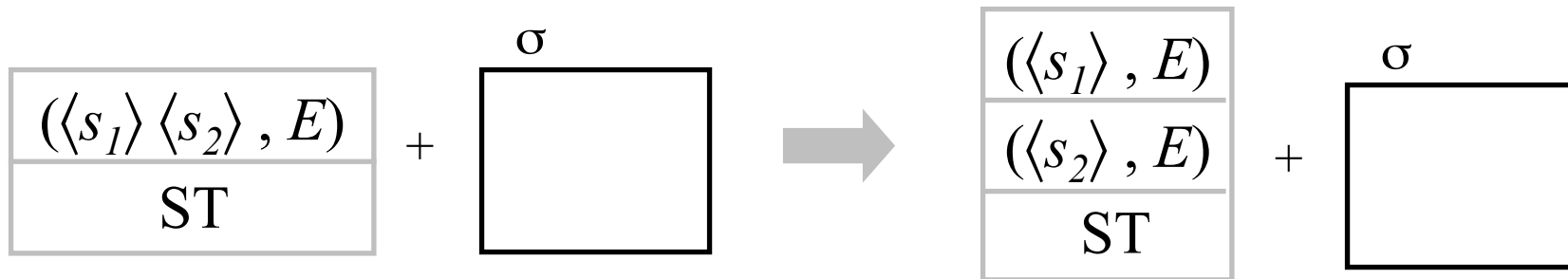
skip

- The semantic statement is
 (skip, E)
- Continue to next execution step



Sequential composition

- The semantic statement is $(\langle s_1 \rangle \langle s_2 \rangle, E)$
- Push $(\langle s_2 \rangle, E)$ and then push $(\langle s_1 \rangle, E)$ on ST
- Continue to next execution step

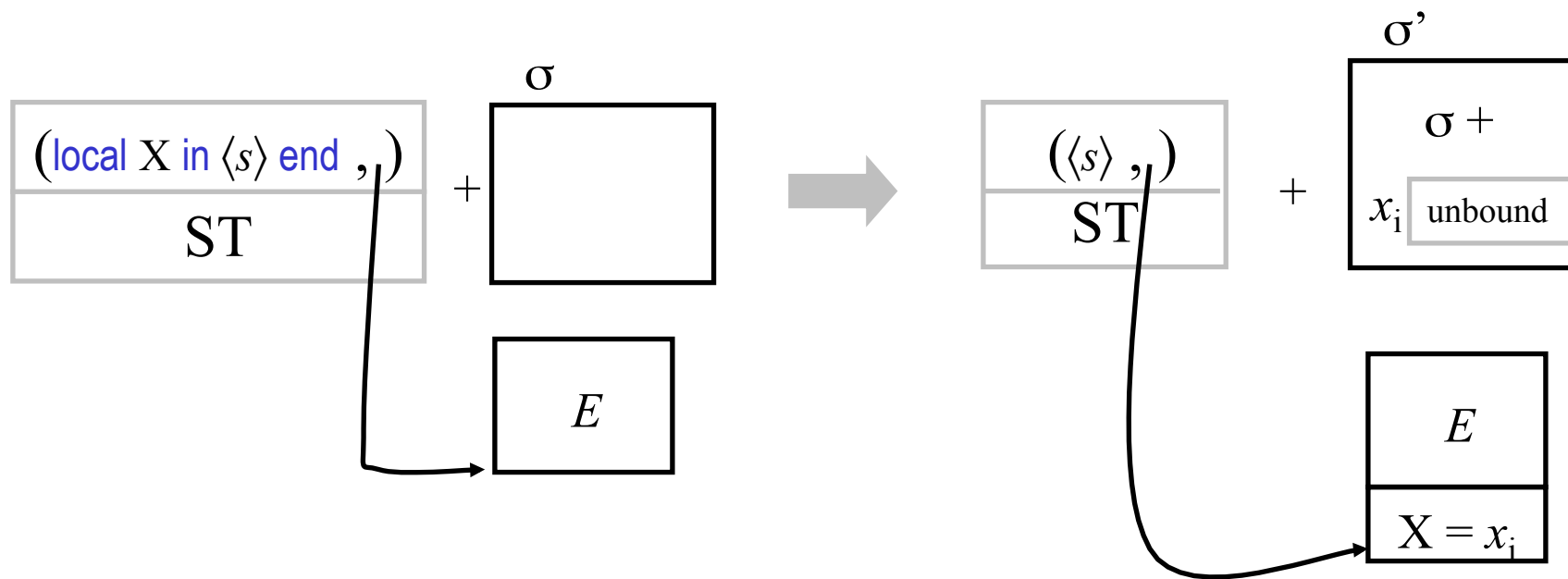


Variable declaration

- The semantic statement is
(local $\langle x \rangle$ in $\langle s \rangle$ end, E)
- Create a new store variable x in the Store
- Let E' be $E + \{\langle x \rangle \rightarrow x\}$, i.e. E' is the same as E but the identifier $\langle x \rangle$ is mapped to x .
- Push $(\langle s \rangle, E')$ on ST
- Continue to next execution step

Variable declaration

- The semantic statement is
 $(\text{local } X \text{ in } \langle s \rangle \text{ end}, E)$



Variable-variable equality

- The semantic statement is
 $(\langle x \rangle = \langle y \rangle, E)$
- Bind $E(\langle x \rangle)$ and $E(\langle y \rangle)$ in the store

Variable-value equality

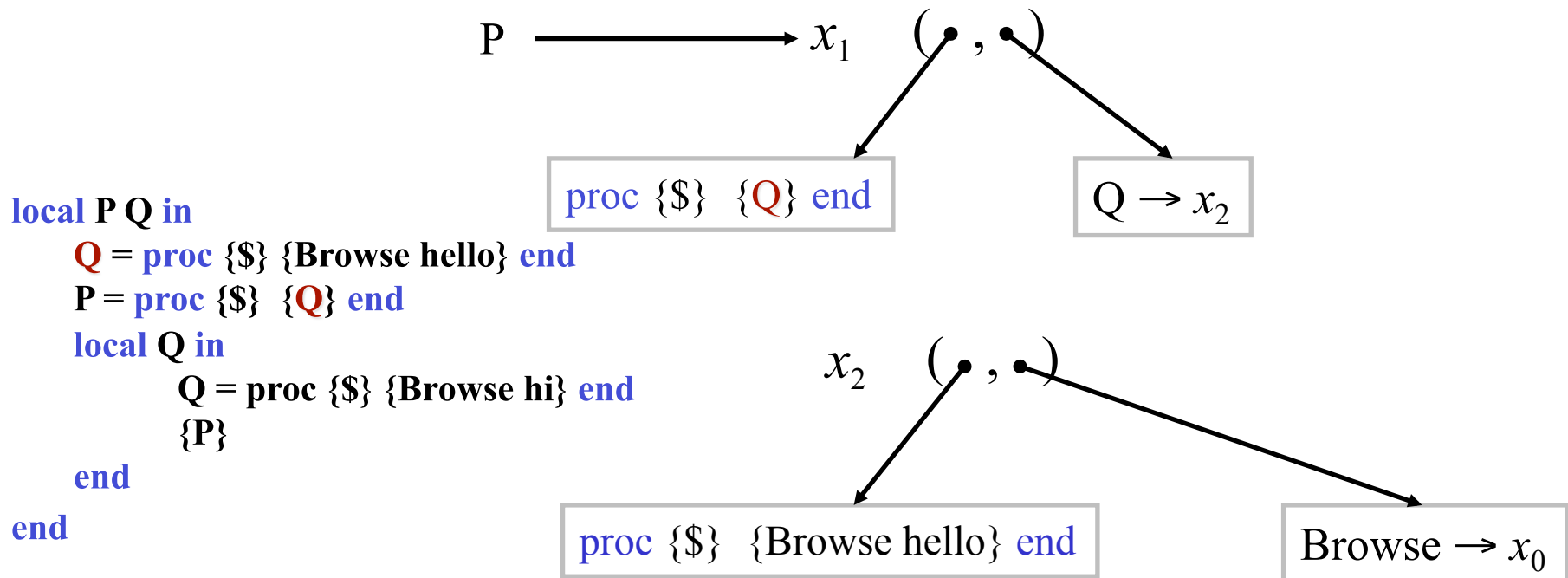
- The semantic statement is
 $(\langle x \rangle = \langle v \rangle, E)$
- Where $\langle v \rangle$ is a record, a number, or a procedure
- Construct the value in the store and refer to it by the variable y .
- Bind $E(\langle x \rangle)$ and y in the store
- We have seen how to construct records and numbers, but what is a procedure value?

Procedure values

- Constructing a procedure value in the store is not simple because a procedure may have external references

```
local P Q in
  Q = proc {$} {Browse hello} end
  P = proc {$} {Q} end
  local Q in
    Q = proc {$} {Browse hi} end
    {P}
  end
end
```

Procedure values (2)



Procedure values (3)

- The semantic statement is
 $(\langle x \rangle = \text{proc } \{\$ \langle y_1 \rangle \dots \langle y_n \rangle\} \langle s \rangle \text{ end}, E)$
- $\langle y_1 \rangle \dots \langle y_n \rangle$ are the (*formal*) parameters of the procedure
- Other free identifiers in $\langle s \rangle$ are called *external references* $\langle z_1 \rangle \dots \langle z_k \rangle$
- These are defined by the environment E where the procedure is declared (lexical scoping)
- The contextual environment of the procedure CE is $E \upharpoonright_{\{\langle z_1 \rangle \dots \langle z_k \rangle\}}$
- When the procedure is called CE is used to construct the environment for execution of $\langle s \rangle$

```
(proc { $  $\langle y_1 \rangle$  ...  $\langle y_n \rangle$  }  
   $\langle s \rangle$   
end ,  
CE)
```

Procedure values (4)

- Procedure values are pairs:
(`proc` {\$ $\langle y_1 \rangle$... $\langle y_n \rangle$ } $\langle s \rangle$ `end` , *CE*)
- They are stored in the store just as any other value

```
(proc {$  $\langle y_1 \rangle$  ...  $\langle y_n \rangle$  }  
   $\langle s \rangle$   
end ,  
CE)
```


Procedure introduction

- The semantic statement is

$$(\langle x \rangle = \text{proc } \{\$ \langle y_1 \rangle \dots \langle y_n \rangle\} \langle s \rangle \text{ end}, E)$$

- Create a contextual environment:

$$CE = E \upharpoonright_{\{\langle z_1 \rangle \dots \langle z_k \rangle\}} \text{ where } \langle z_1 \rangle \dots \langle z_k \rangle \text{ are external references in } \langle s \rangle.$$

- Create a new procedure value of the form:

$$(\text{proc } \{\$ \langle y_1 \rangle \dots \langle y_n \rangle\} \langle s \rangle \text{ end}, CE), \text{ refer to it by the variable } x_p$$

- Bind the store variable $E(\langle x \rangle)$ to x_p
- Continue to next execution step

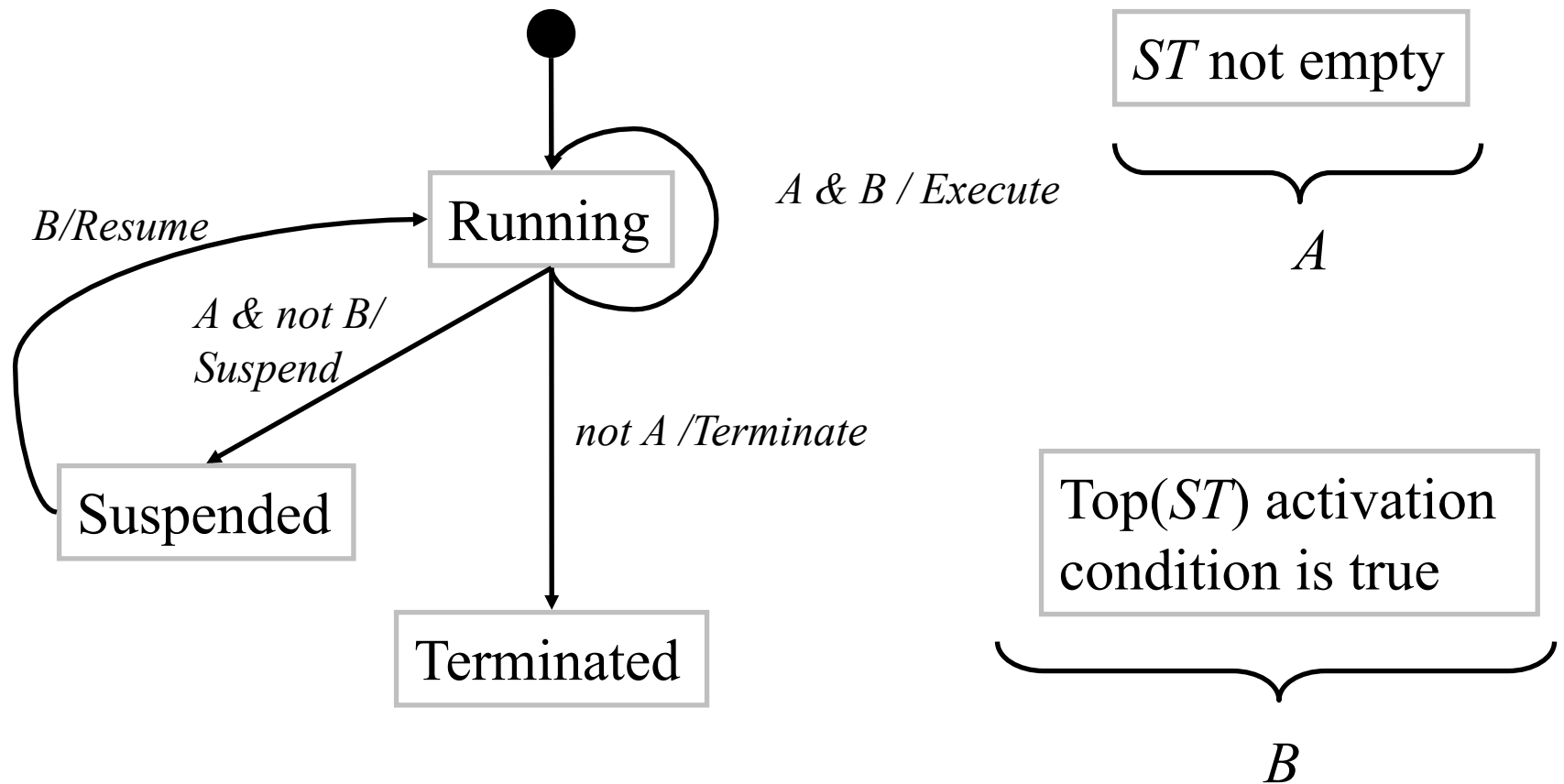
Suspendable statements

- The remaining statements require $\langle x \rangle$ to be bound in order to execute
- The activation condition ($E(\langle x \rangle)$ is *determined*), is that $\langle x \rangle$ be bound to a number, a record or a procedure value

$\langle s \rangle ::=$...

	<code>if $\langle x \rangle$ then $\langle s_1 \rangle$ else $\langle s_2 \rangle$ end</code>	<i>conditional</i>
	<code>{ $\langle x \rangle$ $\langle y_1 \rangle$... $\langle y_n \rangle$ }</code>	<i>procedural application</i>
	<code>case $\langle x \rangle$ of</code> <code> \langlepattern\rangle then $\langle s_1 \rangle$</code> <code>else $\langle s_2 \rangle$ end</code>	<i>pattern matching</i>

Life cycle of a thread



Conditional

- The semantic statement is
(if $\langle x \rangle$ then $\langle s_1 \rangle$ else $\langle s_2 \rangle$ end , E)
- If the activation condition ($E(\langle x \rangle)$ is determined) is true:
 - If $E(\langle x \rangle)$ is not Boolean (true, false), raise an error
 - $E(\langle x \rangle)$ is true, push ($\langle s_1 \rangle$, E) on the stack
 - $E(\langle x \rangle)$ is false, push ($\langle s_2 \rangle$, E) on the stack
- If the activation condition ($E(\langle x \rangle)$ is determined) is false:
 - Suspend

Procedure application

- The semantic statement is $(\{ \langle x \rangle \langle y_1 \rangle \dots \langle y_n \rangle \}, E)$
- If the activation condition ($E(\langle x \rangle)$ is determined) is true:
 - If $E(\langle x \rangle)$ is not a procedure value, or it is a procedure with arity that is not equal to n , raise an error
 - If $E(\langle x \rangle)$ is $(\text{proc } \{ \$ \langle z_1 \rangle \dots \langle z_n \rangle \} \langle s \rangle \text{ end}, CE)$,
push
 $(\langle s \rangle, CE + \{ \langle z_1 \rangle \rightarrow E(\langle y_1 \rangle) \dots \langle z_n \rangle \rightarrow E(\langle y_n \rangle) \})$
on the stack
- If the activation condition ($E(\langle x \rangle)$ is determined) is false:
 - Suspend

Case statement

- The semantic statement is
(**case** $\langle x \rangle$ **of** $\langle l \rangle$ ($\langle f_1 \rangle : \langle x_1 \rangle \dots \langle f_n \rangle : \langle x_n \rangle$)
 then $\langle s_1 \rangle$
 else $\langle s_2 \rangle$ **end** , E)
- If the activation condition ($E(\langle x \rangle)$ is determined) is true:
 - If $E(\langle x \rangle)$ is a record, and the label of $E(\langle x \rangle)$ is $\langle l \rangle$ and its arity is $[\langle f_1 \rangle \dots \langle f_n \rangle]$:
 push (**local** $\langle x_1 \rangle = \langle x \rangle. \langle f_1 \rangle \dots \langle x_n \rangle = \langle x \rangle. \langle f_n \rangle$ **in** $\langle s_1 \rangle$ **end**, E)
 on the stack
 - Otherwise, push ($\langle s_2 \rangle$, E) on the stack
- If the activation condition ($E(\langle x \rangle)$ is determined) is false:
 - Suspend

Execution examples

```

    local Max C in
      proc {Max X Y Z}
        <s>3 if X >= Y then Z=X else Z=Y end
      end
      {Max 3 5 C}
    end
  <s>2
<s>1

```

Execution examples (2)

```

    local Max C in
      proc {Max X Y Z}
        <s>3 if X >= Y then Z=X else Z=Y end
      end
      <s>4 {Max 3 5 C}
    end
  <s>2
<s>1
  
```

- Initial state ($[(\langle s \rangle_1, \emptyset)], \emptyset$)
- After **local Max C in ...**
 $([(\langle s \rangle_2, \{\text{Max} \rightarrow m, C \rightarrow c\})], \{m, c\})$
- After Max binding
 $([(\langle s \rangle_4, \{\text{Max} \rightarrow m, C \rightarrow c\})], \{m = (\text{proc}\{\$ X Y Z\} \langle s \rangle_3 \text{end}, \emptyset), c\})$

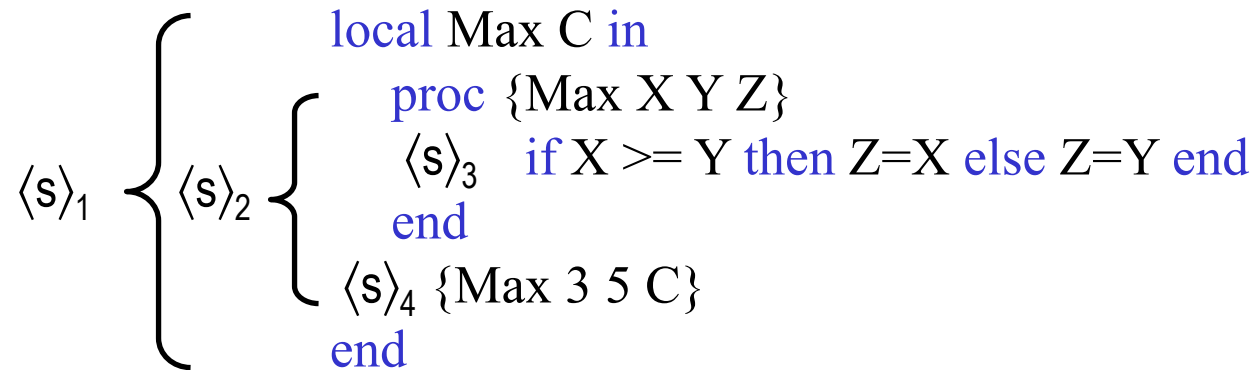
Execution examples (3)

```

    local Max C in
      proc {Max X Y Z}
        <s>3 if X >= Y then Z=X else Z=Y end
      end
      <s>4 {Max 3 5 C}
    end
  <s>2
<s>1
  
```

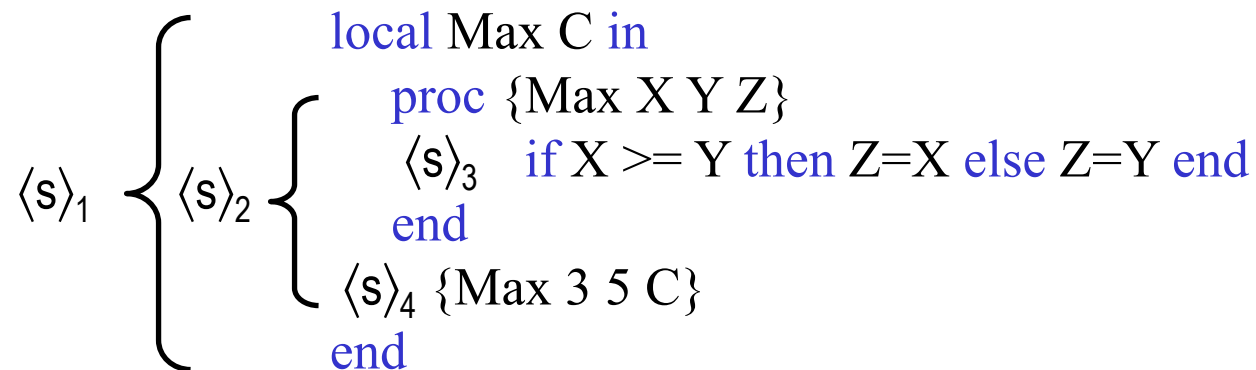
- After Max binding
 $([(\langle s \rangle_4, \{ \text{Max} \rightarrow m, C \rightarrow c \})], \{ m = (\text{proc } \{ \$ X Y Z \} \langle s \rangle_3 \text{end}, \emptyset), c \})$
- After procedure call
 $([(\langle s \rangle_3, \{ X \rightarrow t_1, Y \rightarrow t_2, Z \rightarrow c \})], \{ m = (\text{proc } \{ \$ X Y Z \} \langle s \rangle_3 \text{end}, \emptyset), t_1=3, t_2=5, c \})$

Execution examples (4)



- After procedure call
 $([(\langle s \rangle_3, \{X \rightarrow t_1, Y \rightarrow t_2, Z \rightarrow c\})],$
 $\{m = (\text{proc}\{\$ X Y Z\} \langle s \rangle_3 \text{end}, \emptyset), t_1=3, t_2=5, c\})$
- After $T = (X \geq Y)$
 $([(\langle s \rangle_3, \{X \rightarrow t_1, Y \rightarrow t_2, Z \rightarrow c, T \rightarrow t\})],$
 $\{m = (\text{proc}\{\$ X Y Z\} \langle s \rangle_3 \text{end}, \emptyset), t_1=3, t_2=5, c, t=\text{false}\})$
- $([(Z=Y, \{X \rightarrow t_1, Y \rightarrow t_2, Z \rightarrow c, T \rightarrow t\})],$
 $\{m = (\text{proc}\{\$ X Y Z\} \langle s \rangle_3 \text{end}, \emptyset), t_1=3, t_2=5, c, t=\text{false}\})$

Execution examples (5)



- ([(Z=Y , {X → t₁, Y → t₂, Z → c, T → t})],
 {m = (proc {\$ X Y Z} $\langle s \rangle_3$ end , ∅) , t₁=3, t₂=5, c, t=false})
- ([],
 {m = (proc {\$ X Y Z} $\langle s \rangle_3$ end , ∅) , t₁=3, t₂=5, c=5, t=false})

Exercises

46. Does dynamic binding require keeping an environment in a closure (procedure value)? Why or why not?
47. VRH Exercise 2.9.2 (page 107)
48. After translating the following function to the kernel language:

```
fun {AddList L1 L2}
  case L1 of H1|T1 then
    case L2 of H2|T2 then
      H1+H2|{AddList T1 T2}
    end
  else nil end
end
```

Use the operational semantics to execute the call

```
{AddList [1 2] [3 4]}
```