Declarative Computation Model

Kernel language semantics

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

Carlos Varela

RPI

March 18, 2010

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, ⟨s⟩ denotes a statement

⟨s⟩	::=	skip	*empty statement*
	|	⟨x⟩ = ⟨y⟩	*variable-variable binding*
	|	⟨x⟩ = ⟨v⟩	*variable-value binding*
	|	$\langle s_1 \rangle$ $\langle s_2 \rangle$	*sequential composition*
	|	local ⟨x⟩ in $\langle s_1 \rangle$ end	*declaration*
	|	if ⟨x⟩ then $\langle s_1 \rangle$ else $\langle s_2 \rangle$ end	*conditional*
	|	{ ⟨x⟩ $\langle y_1 \rangle$ … $\langle y_n \rangle$ }	*procedural application*
	|	case ⟨x⟩ of ⟨pattern⟩ then $\langle s_1 \rangle$ else $\langle s_2 \rangle$ end	*pattern matching*

⟨v⟩	::=	proc { \$ $\langle y_1 \rangle$ … $\langle y_n \rangle$ } $\langle s_1 \rangle$ end | ...	*value expression*
⟨pattern⟩	::=	...	

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

  (⟨s⟩ , *E*) where ⟨s⟩ 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)

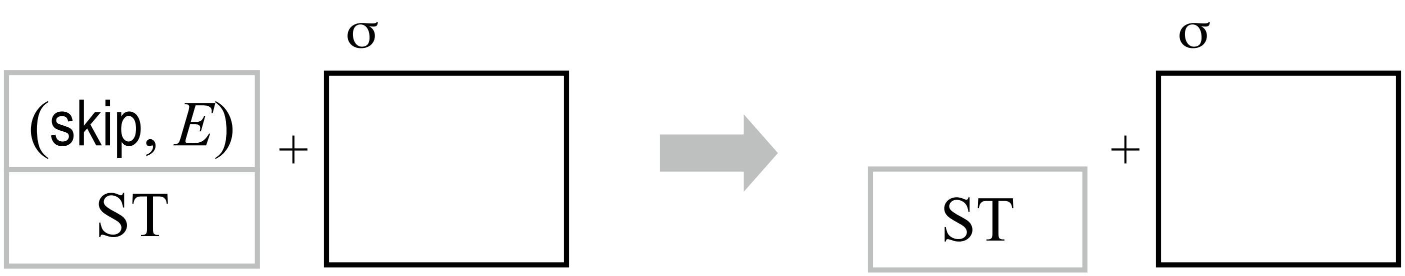
  – where ST is a *stack of semantic statements* and σ is a *single assignment store*
- A *computation* is a sequence of execution states
  $(ST_0 , \sigma_0) \rightarrow (ST_1 , \sigma_1) \rightarrow (ST_2 , \sigma_2) \rightarrow \ldots$

Semantics

- To execute a program (i.e., a statement) $\langle s \rangle$ the initial execution state is
  $$([(\langle s \rangle , \varnothing)] , \varnothing)$$
- *ST* has a single semantic statement $(\langle s \rangle , \varnothing)$
- The environment *E* is empty, and the store σ is empty
- [...] 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

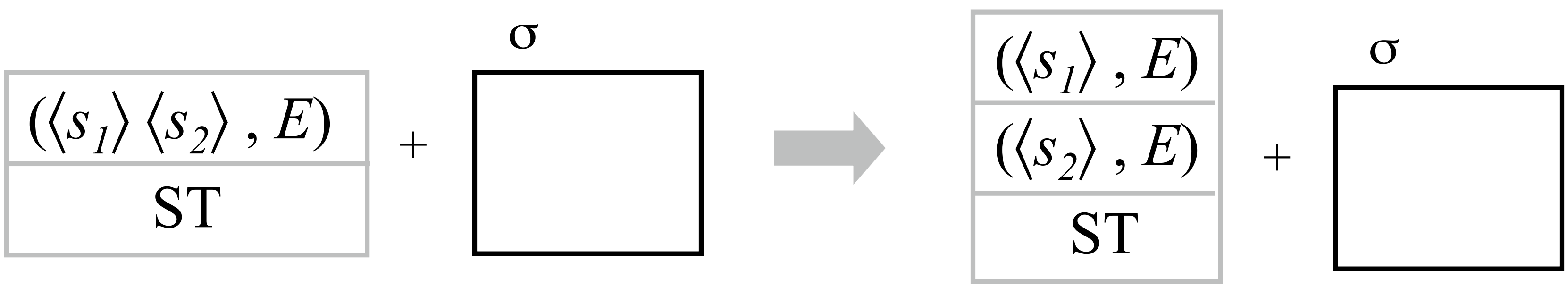
skip

- 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



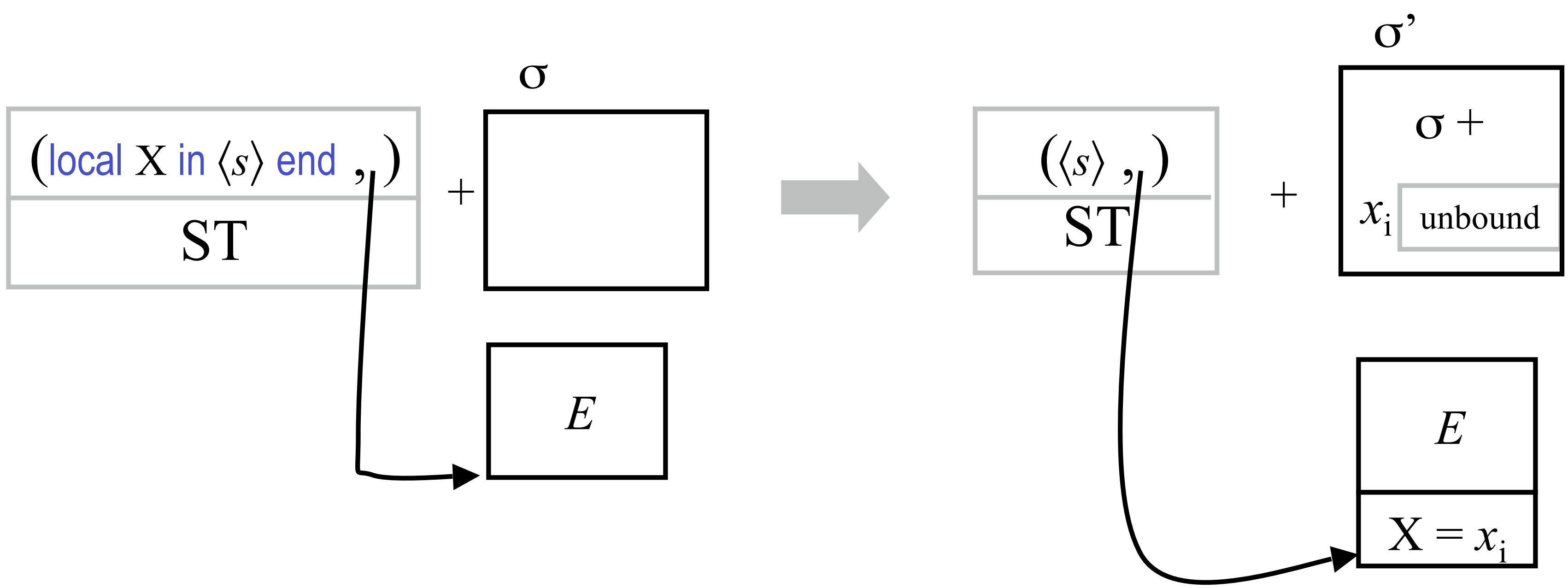
Variable declaration

- The semantic statement is

  (local ⟨x⟩ in ⟨*s*⟩ end, *E*)
- Create a new store variable *x* in the Store
- Let *E'* be *E*+{⟨x⟩ → *x*}, i.e. *E'* is the same as *E* but the identifier ⟨*x*⟩ is mapped to *x*.
- Push (⟨*s*⟩ , *E'*) on *ST*
- Continue to next execution step

Variable declaration

- The semantic statement is
  (local X in ⟨*s*⟩ 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

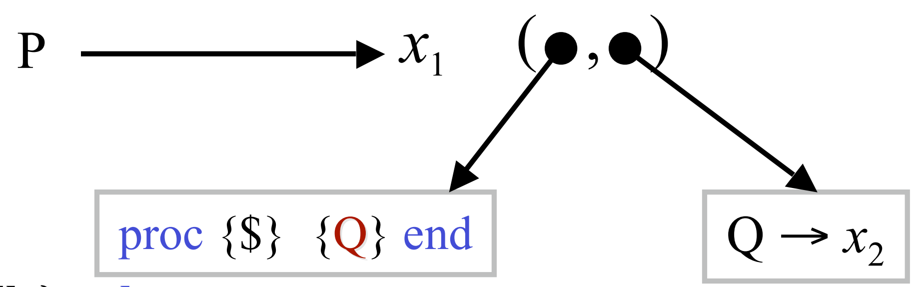
- The semantic statement is
  $(\langle x \rangle = \langle v \rangle, E)$
- Where $\langle v \rangle$ is a record, a number, or a procedure
- Construct the value in the store and refer to it by the variable y.
- Bind $E(\langle x \rangle)$ and y in the store
- We have seen how to construct records and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

Procedure values

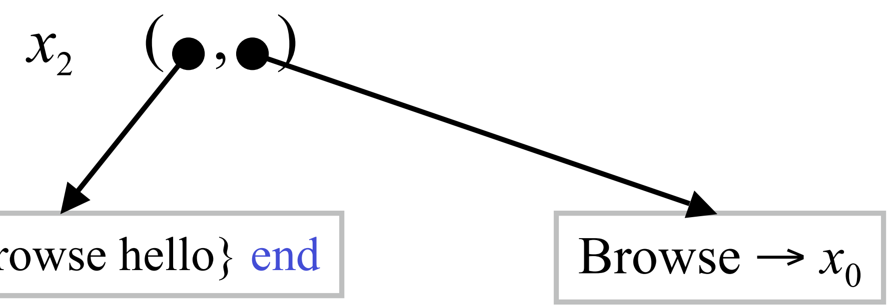
- Constructing a procedure value in the store is not simple because a procedure may have external references

```
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)




```
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 (3)

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

```
(proc {$ ⟨y1⟩ … ⟨yn⟩ }
    ⟨s⟩
 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 {$ ⟨y1⟩ … ⟨yn⟩ }
    ⟨s⟩
 end ,
CE)
```

Procedure introduction

- The semantic statement is

  $(\langle x \rangle = \texttt{proc}\ \{\$\ \langle y_1 \rangle \ldots \langle y_n \rangle\}\ \langle s \rangle\ \texttt{end}, E)$
- Create a contextual environment:

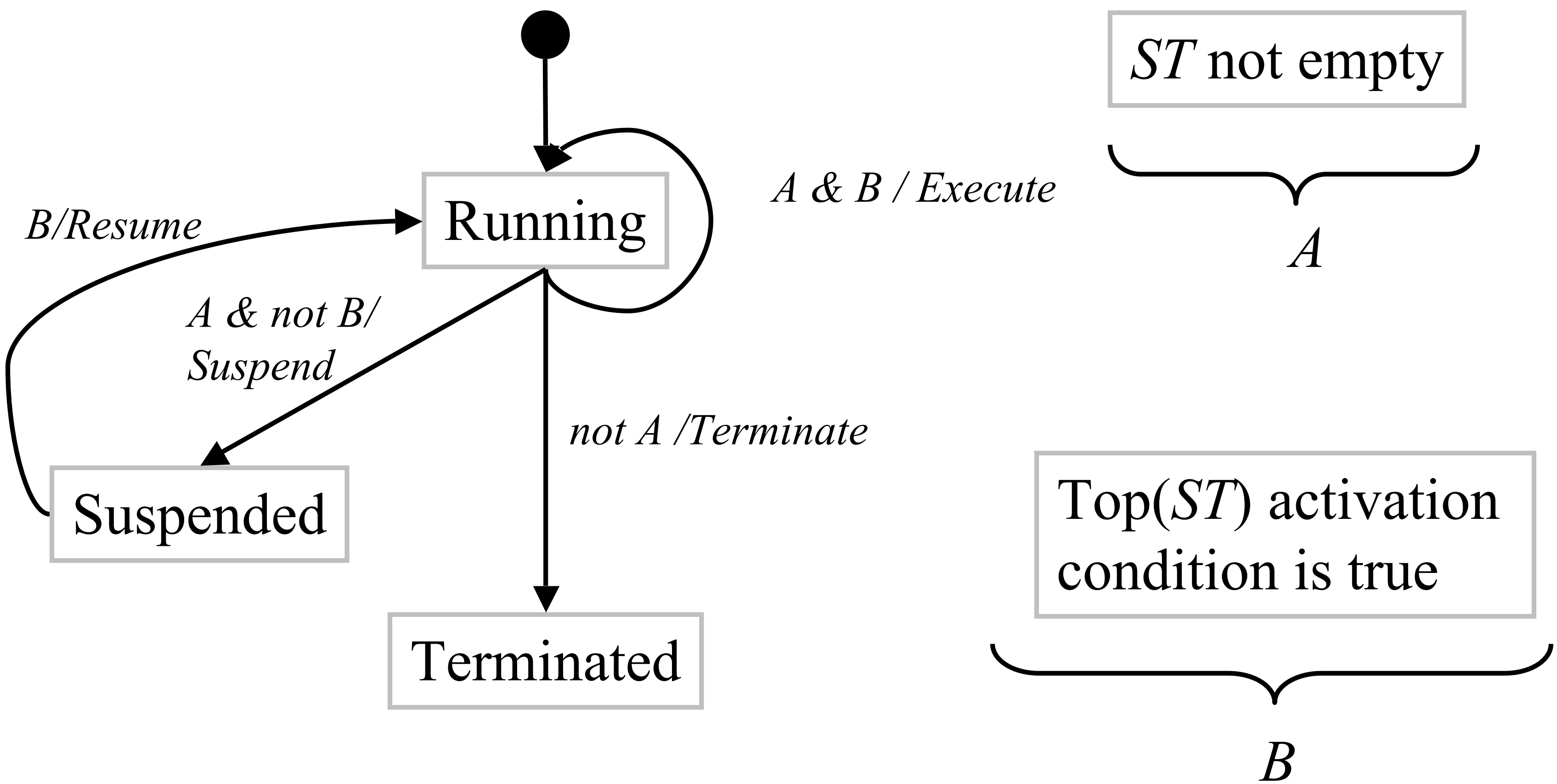
  $CE = E\,|_{\{\langle z_1 \rangle \ldots \langle z_k \rangle\}}$ where $\langle z_1 \rangle \ldots \langle z_k \rangle$ are external references in $\langle s \rangle$.
- Create a new procedure value of the form: $(\texttt{proc}\ \{\$\ \langle y_1 \rangle \ldots \langle y_n \rangle\}\ \langle s \rangle\ \texttt{end}, CE)$, 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 ⟨x⟩ to be bound in order to execute
- The activation condition (*E*(⟨x⟩) is *determined*), is that ⟨x⟩ be bound to a number, a record or a procedure value

⟨s⟩ ::= …	
| if ⟨x⟩ then $\langle s_1 \rangle$ else $\langle s_2 \rangle$ end	*conditional*
| { ⟨x⟩ $\langle y_1 \rangle$ … $\langle y_n \rangle$ }	*procedural application*
| case ⟨x⟩ of ⟨pattern⟩ then $\langle s_1 \rangle$ else $\langle s_2 \rangle$ end	*pattern matching*

Life cycle of a thread



Conditional

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

Procedure application

- The semantic statement is
  $(\{ \langle x \rangle \langle y_1 \rangle \ldots \langle y_n \rangle \}, E)$
- 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 (proc {$ $\langle z_1 \rangle \ldots \langle z_n \rangle$} $\langle s \rangle$ end, CE) ,
    push
    $(\langle s \rangle , CE + \{\langle z_1 \rangle \rightarrow E(\langle y_1 \rangle) \ldots \langle z_n \rangle \rightarrow E(\langle y_n \rangle)\})$
    on the stack
- If the activation condition ($E(\langle x \rangle)$ is determined) is false:
  - Suspend

Case statement

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

Execution examples

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

⟨s⟩1 spans from `local Max C in` to the final `end`; ⟨s⟩2 spans `proc {Max X Y Z}` … `end`; ⟨s⟩3 is `if X >= Y then Z=X else Z=Y end`.

Execution examples (2)

$\langle s \rangle_1$ { local Max C in
  $\langle s \rangle_2$ { proc {Max X Y Z}
    $\langle s \rangle_3$ if X >= Y then Z=X else Z=Y end
    end
  $\langle s \rangle_4$ {Max 3 5 C}
end

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

Execution examples (3)

$\langle s \rangle_1$ $\langle s \rangle_2$

```
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
```

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

Execution examples (4)

$\langle s \rangle_1$ { local Max C in
  $\langle s \rangle_2$ { proc {Max X Y Z}
    $\langle s \rangle_3$ if X >= Y then Z=X else Z=Y end
    end
  $\langle s \rangle_4$ {Max 3 5 C}
end

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

Execution examples (5)

$\langle s \rangle_1$ { local Max C in
$\langle s \rangle_2$ { proc {Max X Y Z}
$\langle s \rangle_3$ if X >= Y then Z=X else Z=Y end
end
$\langle s \rangle_4$ {Max 3 5 C}
end

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