Declarative Computation Model

Kernel language semantics

(Non-)Suspendable statements (CTM 2.4.3-2.4.5)

Carlos Varela

RPI

March 12, 2015

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
  $(\langle s \rangle , E)$ 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)

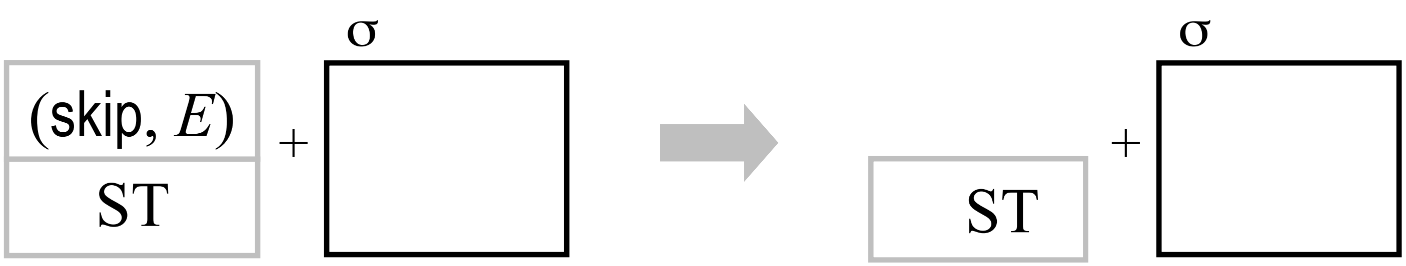
  - where ST is a *stack of semantic statements* and σ is a *single assignment store*
- A *computation* is a sequence of execution states
  $(ST_0 , \sigma_0) \rightarrow (ST_1 , \sigma_1) \rightarrow (ST_2 , \sigma_2) \rightarrow \ldots$

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

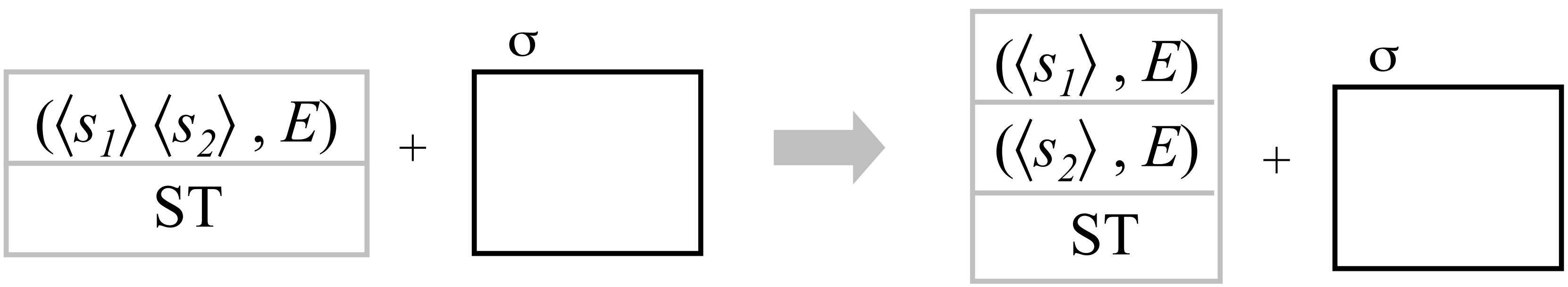
- The semantic statement is
  (skip, E)
- Continue to next execution step



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

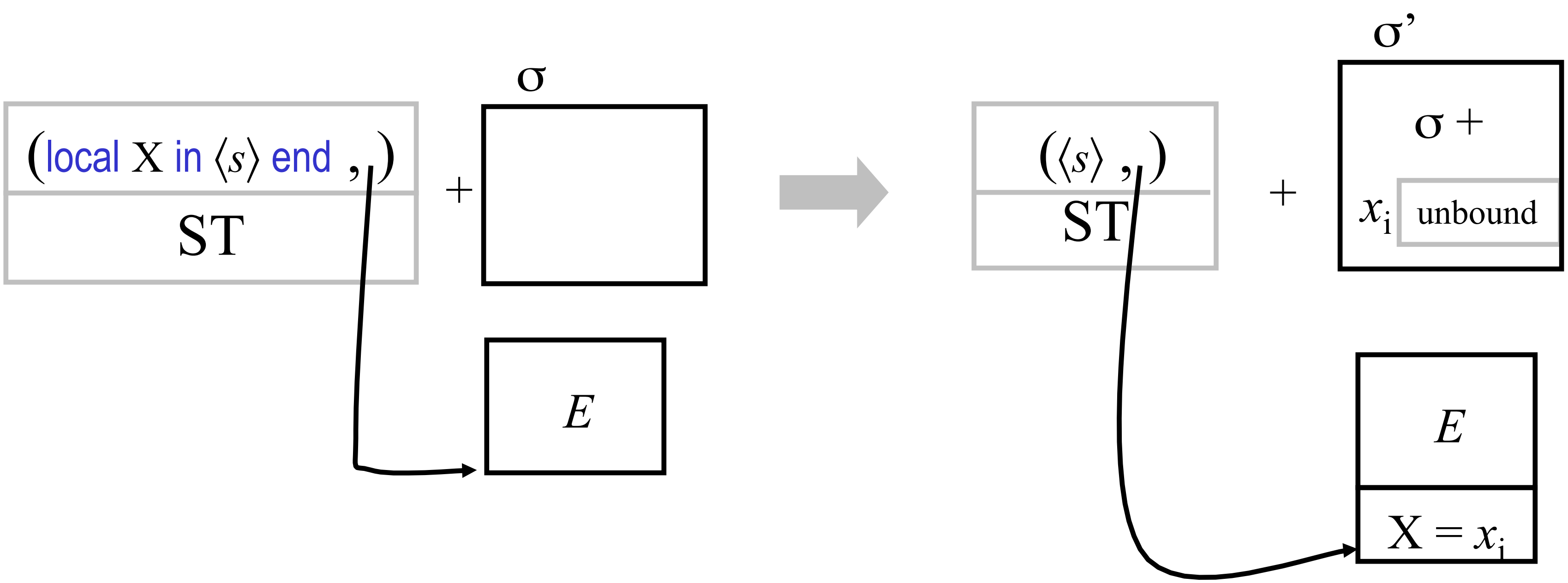


Variable declaration

- The semantic statement is
  (local $\langle x \rangle$ 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
  (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

  $(\langle x \rangle = \langle v \rangle, E)$

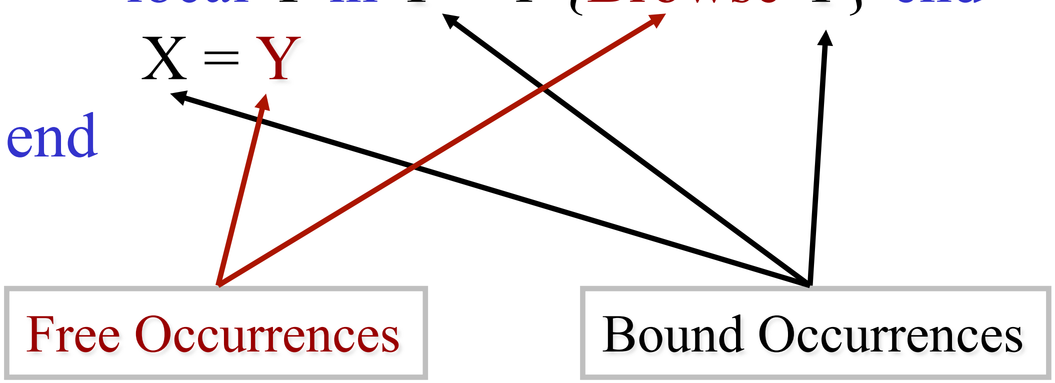
- Where $\langle v \rangle$ is a record, a number, or a procedure
- Construct the value in the store and refer to it by the variable *y*.
- Bind $E(\langle x \rangle)$ and *y* in the store
- We have seen how to construct records and numbers, but what is a procedure value?

Lexical scoping

- Free and bound identifier occurrences
- An identifier occurrence is *bound* with respect to a statement ⟨s⟩ if it is in the scope of a declaration inside ⟨s⟩
- A variable identifier is declared either by a 'local' statement, as a parameter of a procedure, or implicitly declared by a case statement
- An identifier occurrence is *free* otherwise
- In a running program every identifier is bound (i.e., declared)

Lexical scoping (2)

- ```
  proc {P X}
      local Y in Y = 1 {Browse Y} end
      X = Y
  end
  ```


  ```

Lexical scoping (3)

- local Arg1 Arg2 in

```
    Arg1 = 111*111
    Arg2 = 999*999
    Res = Arg1*Arg2
end
```

Free Occurrences

Bound Occurrences

This is not a runnable program!

Lexical scoping (4)

```
• local Res in
      local Arg1 Arg2 in
            Arg1 = 111*111
            Arg2 = 999*999
            Res = Arg1*Arg2
      end
      {Browse Res}
  end
```

Lexical scoping (5)

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

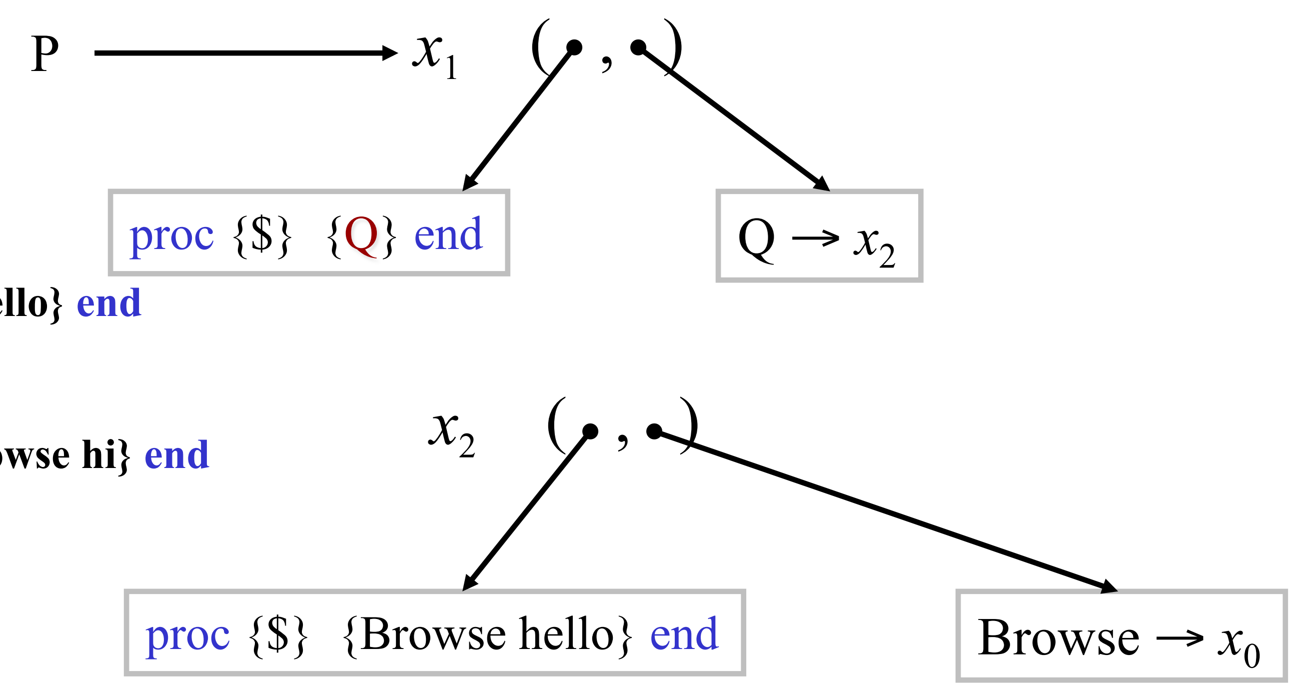
Procedure values

- 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 {\$ $\langle y_1 \rangle \ldots \langle y_n \rangle$ }
    $\langle s \rangle$
 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

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

- 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 $\langle s \rangle$.

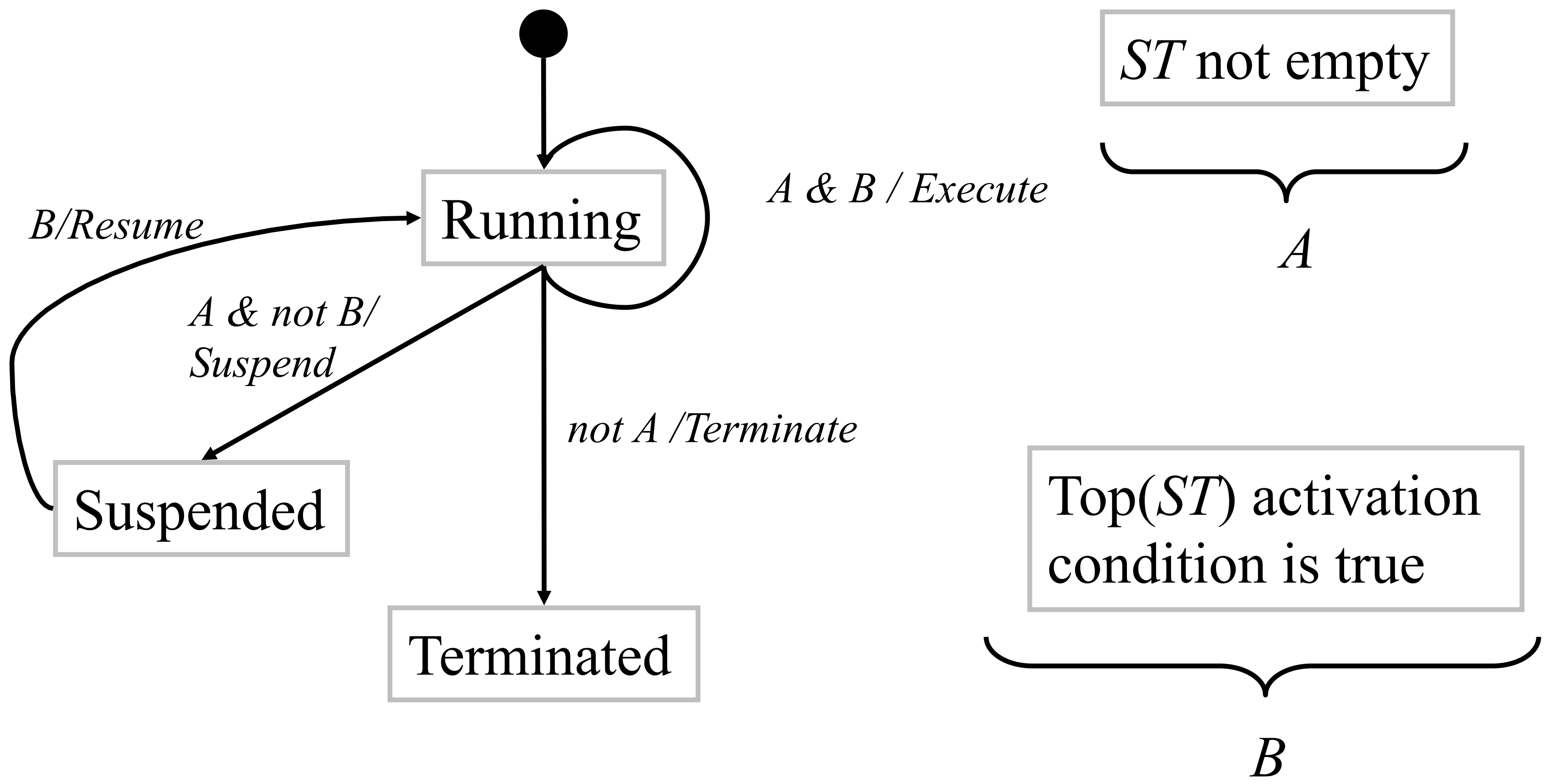
- Create a new procedure value of the form: $(\text{proc } \{\$ \ \langle y_1 \rangle \ \ldots \ \langle y_n \rangle\} \ \langle s \rangle \text{ end}, CE)$, refer to it by the variable x_P
- Bind the store variable $E(\langle x \rangle)$ to x_P
- Continue to next execution step

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
  $(\{ \langle x \rangle \, \langle y_1 \rangle \dots \langle y_n \rangle \} , E)$
- 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 (proc {$ $\langle z_1 \rangle \dots \langle z_n \rangle$} $\langle s \rangle$ 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 ($E(\langle \mathrm{x} \rangle)$ 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 (⟨s_1⟩, E+{⟨x_1⟩ → E(⟨x⟩).⟨f_1⟩,…, ⟨x_n⟩ → E(⟨x⟩).⟨f_n⟩}) on the stack
  - Otherwise, push (⟨s_2⟩, E) on the stack
- If the activation condition (E(⟨x⟩) is determined) is false:
  - Suspend

Execution example

```
local Max C in
   fun{Max X Y}
         if X >= Y then X else Y end
   end
   C = {Max 3 5}
end
```

Example in kernel language

```
⟨s⟩  local Max in
       local A in
         local B in
           local C in
 ⟨s⟩1        Max = proc {$ X Y Z}
     ⟨s⟩3      local T in
                  T = (X >= Y)
     ⟨s⟩4         if T then Z=X else Z=Y end
               end
             end
             A = 3
             B = 5
 ⟨s⟩2        {Max A B C}
           end
         end
       end
     end
```

Execution example (2)

- Initial state: ([(⟨s⟩, ∅)], ∅)

- After four local … in
  ([(⟨s⟩$_1$, {Max → *m*, A → *a*, B → *b*, C → *c*})], {*m, a, b, c*})

- After bindings of Max, A, B:
  ([(⟨s⟩$_2$, {Max → *m*, A → *a*, B → *b*, C → *c*})],
  {*m* = (proc{$ X Y Z} ⟨s⟩$_3$ end , ∅) , *a = 3, b = 5, c*})

- After {Max A B C} procedure call:
  ([(⟨s⟩$_3$, {X → *a*, Y → *b*, Z → *c*})],
  {*m* = (proc{$ X Y Z} ⟨s⟩$_3$ end , ∅) , *a* = 3, *b* = 5, c})

Execution example (3)

- After T = (X>=Y):

  ([($\langle s \rangle_4$, {X → a, Y → b, Z → c, T → t})],
  {m = (proc{$ X Y Z} $\langle s \rangle_3$ end , ∅) , a = 3, b = 5, c, t = false})

- After the execution of the conditional statement:

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

- After the variable-variable binding:

  ([ ],
  {m = (proc{$ X Y Z} $\langle s \rangle_3$ end , ∅) , a = 3, b = 5, c = 5, t=false})

  We end execution since the statement stack is empty. The store variable c is effectively bound to 5 (*i.e.,* {Max 3 5}) in the store.

Procedures with external references

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

Procedures with external references

$\langle s \rangle_1$ / $\langle s \rangle_2$ / $\langle s \rangle_3$:

```
local LB Y C in
   Y = 5
     proc {LB X Z}
          if X >= Y then Z=X else Z=Y end
     end
     {LB 3  C}
end
```

- The procedure value of LB is
- (proc{$ X Z} $\langle s \rangle_3$ end , {Y → y})
- The store is $\{y = 5, \ldots\}$

Procedures with external references

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

- The procedure value of LB is
- (proc{\$ X Z} ⟨s⟩$_3$ end , {Y $\rightarrow$ y})
- The store is $\{y = 5, \ldots\}$
- STACK: [({LB T C} , {Y $\rightarrow$ y , LB $\rightarrow$ lb, C $\rightarrow$ c, T $\rightarrow$ t})]
- STORE: {y = 5, lb = (proc{\$ X Z} ⟨s⟩$_3$ end , {Y $\rightarrow$ y}) , t = 3, c}

Procedures with external references

$\langle s \rangle_1$ / $\langle s \rangle_2$ / $\langle s \rangle_3$:

```
local LB Y C in
  Y = 5
  proc {LB X Z}
    if X >= Y then Z=X else Z=Y end
  end
  {LB 3 C}
end
```

- STACK: $[(\ \{LB\ T\ C\}\ ,\ \{Y \rightarrow y\ ,\ LB \rightarrow lb,\ C \rightarrow c,\ T \rightarrow t\})\ ]$
- STORE: $\{y = 5,\ lb = (\mathtt{proc}\{\$\ X\ Z\}\ \langle s \rangle_3\ \mathtt{end}\ ,\ \{Y \rightarrow y\})\ ,\ t = 3,\ c\}$
- STACK: $[(\langle s \rangle_3\ ,\ \{Y \rightarrow y\ ,\ X \rightarrow t\ ,\ Z \rightarrow c\})\ ]$
- STORE: $\{y = 5,\ lb = (\mathtt{proc}\{\$\ X\ Z\}\ \langle s \rangle_3\ \mathtt{end}\ ,\ \{Y \rightarrow y\})\ ,\ t = 3,\ c\}$

Procedures with external references

$\langle s \rangle_1$, $\langle s \rangle_2$, $\langle s \rangle_3$:

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

- STACK: $[(\langle s \rangle_3, \{Y \rightarrow y, X \rightarrow t, Z \rightarrow c\})]$
- STORE: $\{y = 5, lb = (\text{proc}\{\$\ X\ Z\}\ \langle s \rangle_3\ \text{end}, \{Y \rightarrow y\}), t = 3, c\}$
- STACK: $[(Z{=}Y, \{Y \rightarrow y, X \rightarrow t, Z \rightarrow c\})]$
- STORE: $\{y = 5, lb = (\text{proc}\{\$\ X\ Z\}\ \langle s \rangle_3\ \text{end}, \{Y \rightarrow y\}), t = 3, c\}$
- STACK: [ ]
- STORE: $\{y = 5, lb = (\text{proc}\{\$\ X\ Z\}\ \langle s \rangle_3\ \text{end}, \{Y \rightarrow y\}), t = 3, c = 5\}$

Try the following!

```
local LB Y C in
      Y = 5
      proc {LB X Z}
            if X >= Y then Z=X else Z=Y end
      end
      local Y in
        Y = 10
        {LB 3 C}
      end
end
```

Exercises

46. Does dynamic binding require keeping an environment in a closure (procedure value)? Why or why not?
47. CTM 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]}
```

Exercises

49. CTM Exercise 2.9.3 (page 107).
50. CTM Exercise 2.9.7 (page 108) –translate example to kernel language and execute using operational semantics.
51. Write an example of a program that suspends. Now, write an example of a program that never terminates. Use the operational semantics to prove suspension or non-termination.