Declarative Computation Model

Kernel language semantics
(Non-)Suspendable statements (VRH 2.4.3-2.4.4)

Carlos Varela
RPI
September 28, 2006

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*
| ⟨s_1⟩ ⟨s_2⟩	*sequential composition*
| local ⟨x⟩ in ⟨s_1⟩ end	*declaration*
| if ⟨x⟩ then ⟨s_1⟩ else ⟨s_2⟩ end	*conditional*
| { ⟨x⟩ ⟨y_1⟩ ... ⟨y_n⟩ }	*procedural application*
| case ⟨x⟩ of ⟨pattern⟩ then ⟨s_1⟩ else ⟨s_2⟩ end	*pattern matching*
⟨v⟩ ::= proc { $ ⟨y_1⟩ ... ⟨y_n⟩ } ⟨s_1⟩ 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 → x_1, Y → x_2}
- A *semantic statement* is a pair
  (⟨s⟩ , *E*) where ⟨s⟩ is a statement
- *ST* is a stack of <u>semantic statements</u>

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 pair
  (*ST* , σ)
- *ST* is a stack of <u>semantic statements</u>
- A *computation* is a sequence of execution states
  $(ST_0 , \sigma_0) \rightarrow (ST_1 , \sigma_1) \rightarrow (ST_2 , \sigma_2) \rightarrow ...$

Semantics

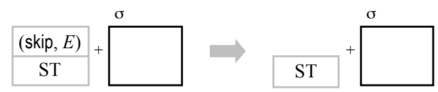
- To execute a program (i.e., a statement) ⟨*s*⟩ the initial execution state is
  ([(⟨*s*⟩ , ∅)] , ∅)
- *ST* has a single semantic statement (⟨*s*⟩ , ∅)
- The environment *E* is empty, and the store σ is empty
- [...] denotes the stack
- At each step the first element of *ST* is popped and execution 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

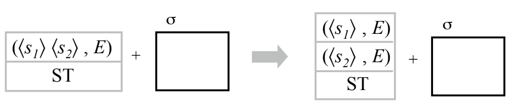
skip

- 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

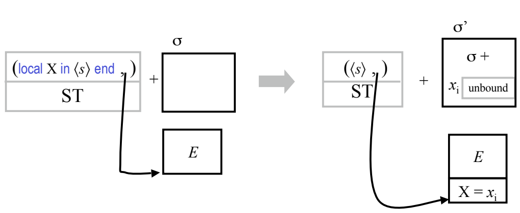


Variable declaration

- 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 $\langle s \rangle$ 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?

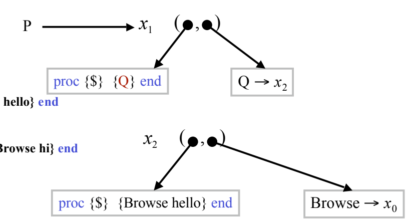
Procedure values

- 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
  (proc {\$ $\langle y_1\rangle$ ... $\langle y_n\rangle$} $\langle s\rangle$ end, E)
- $\langle y_1\rangle$ ... $\langle y_n\rangle$ are the (formal) parameters of the procedure
- Other free identifiers of $\langle s\rangle$ are called external references $\langle z_1\rangle$ ... $\langle z_k\rangle$
- 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 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

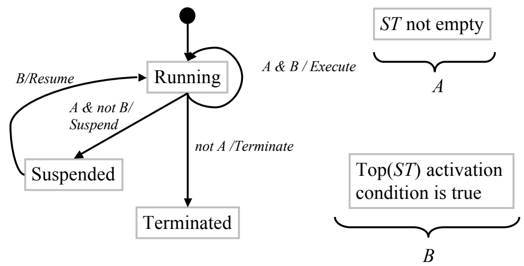
- The semantic statement is ($\langle x\rangle$ = proc {\$ $\langle y_1\rangle$ ... $\langle y_n\rangle$} $\langle s\rangle$ end, E)
- Environment is $\{\langle x\rangle \rightarrow x_P, \ldots\}$
- Create a new procedure value of the form:
  (proc {\$ $\langle y_1\rangle$ ... $\langle y_n\rangle$} $\langle s\rangle$ 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 <u>activation condition</u> (*E*(⟨x⟩) is *<u>determined</u>*), i.e., ⟨x⟩ is bound to a number, record or a procedure value

```
⟨s⟩ ::= ...
    |   if ⟨x⟩ then ⟨s1⟩ else ⟨s2⟩ end        conditional
    |   { ⟨x⟩ ⟨y1⟩ … ⟨yn⟩ }                    procedural application
    |   case ⟨x⟩ of                          pattern matching
            ⟨pattern⟩ then ⟨s1⟩
        else ⟨s2⟩ end
```


Life cycle of a thread



Conditional

- The semantic statement is
  (if ⟨x⟩ then ⟨s_1⟩ else ⟨s_2⟩ end , *E*)
- 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
- The activation condition (*E*(⟨x⟩) is determined) is false: suspend

Procedure application

- The semantic statement is
  ({ ⟨*x*⟩ ⟨y_1⟩ … ⟨y_n⟩ } , *E*)
- The activation condition (*E*(⟨*x*⟩) is determined) is true:
  - If *E*(⟨*x*⟩) is not procedure value, or a procedure with arity that is not equal to *n*, raise an error
  - *E*(⟨*x*⟩) is (proc {$ ⟨z_1⟩ ... ⟨z_n⟩} ⟨*s*⟩ end, *CE*) , push
    (⟨*s*⟩ , *CE* + {⟨z_1⟩ → *E*(⟨y_1⟩) … ⟨z_n⟩ → *E*(⟨y_n⟩)})
    on the stack
- The activation condition (*E*(⟨x⟩) 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*)
- The activation condition (*E*(⟨x⟩) is determined) is true:
  - 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
- The activation condition (*E*(⟨x⟩) is determined) is false: suspend

Execution examples

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

Execution examples (2)

```
⟨s⟩1 {        local Max C in
      ⟨s⟩2 {     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
```

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

Execution examples (3)

```
⟨s⟩1 {        local Max C in
      ⟨s⟩2 {     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
  $([(\langle s\rangle_4, \{\text{Max} \rightarrow m, \text{C} \rightarrow c\})],$
  $\{m = (\text{proc}\{\$ \text{ X Y Z}\}\ \langle s\rangle_3 \text{end}, \varnothing), c\})$
- After procedure call
  $([(\langle s\rangle_3, \{\text{X} \rightarrow t_1, \text{Y} \rightarrow t_2, \text{Z} \rightarrow c\})],$
  $\{m = (\text{proc}\{\$ \text{ X Y Z}\}\ \langle s\rangle_3 \text{end}, \varnothing), t_1{=}3, t_2{=}5, c\})$

Execution examples (4)

```
⟨s⟩1 {        local Max C in
      ⟨s⟩2 {     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 procedure call
  $([(\langle s\rangle_3, \{\text{X} \rightarrow t_1, \text{Y} \rightarrow t_2, \text{Z} \rightarrow c\})],$
  $\{m = (\text{proc}\{\$ \text{ X Y Z}\}\ \langle s\rangle_3 \text{end}, \varnothing), t_1{=}3, t_2{=}5, c\})$
- After T = (X>=Y)
  $([(\langle s\rangle_3, \{\text{X} \rightarrow t_1, \text{Y} \rightarrow t_2, \text{Z} \rightarrow c, \text{T} \rightarrow t\})],$
  $\{m = (\text{proc}\{\$ \text{ X Y Z}\}\ \langle s\rangle_3 \text{end}, \varnothing), t_1{=}3, t_2{=}5, c, t{=}\text{false}\})$
- $([(\text{Z=Y}, \{\text{X} \rightarrow t_1, \text{Y} \rightarrow t_2, \text{Z} \rightarrow c, \text{T} \rightarrow t\})],$
  $\{m = (\text{proc}\{\$ \text{ X Y Z}\}\ \langle s\rangle_3 \text{end}, \varnothing), t_1{=}3, t_2{=}5, c, t{=}\text{false}\})$

Execution examples (5)

```
⟨s⟩1 {        local Max C in
      ⟨s⟩2 {     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
```

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

Exercises

31. Does dynamic binding require keeping an environment in a closure (procedure value)? Why or why not?
32. VRH Exercise 2.9.2 (page 107)
33. *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]}
```