Declarative Computation Model

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

Carlos Varela
RPI

October 11, 2007

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)

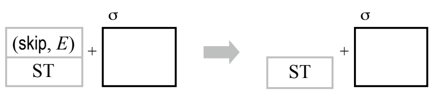
- A computation defines how the execution state is transformed step by step from the initial state to the final state
- The *execution state* is a pair
  (*ST* , σ)
  - where *ST* is a *stack of semantic statements* and σ is a *single assignment store*
- A *computation* is a sequence of execution states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

Semantics

- To execute a program (i.e., a statement) ⟨*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

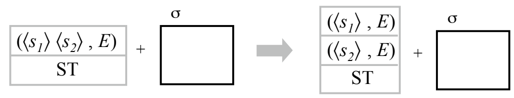
skip

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



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

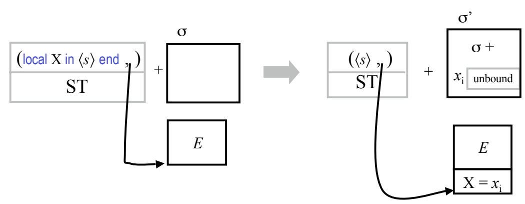


Variable declaration

- The semantic statement is
  (local ⟨x⟩ in ⟨s⟩ end, E)
- Create a new store variable x in the Store
- Let E' be $E+\{\langle x\rangle \rightarrow x\}$, i.e. E' is the same as E but the identifier ⟨x⟩ 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
  (⟨x⟩ = ⟨y⟩, E)
- Bind $E(\langle x\rangle)$ and $E(\langle y\rangle)$ in the store

Variable-value equality

- The semantic statement is
  (⟨x⟩ = ⟨v⟩, E)
- Where ⟨v⟩ 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?

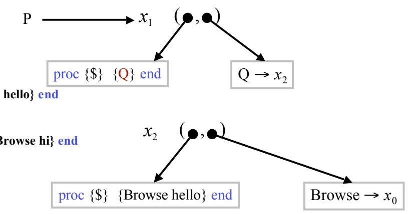
Procedure values

- 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
  (⟨*x*⟩ = proc {$ ⟨y_1⟩ ... ⟨y_n⟩} ⟨*s*⟩ end, *E*)
- ⟨y_1⟩ ... ⟨y_n⟩ are the (*formal*) parameters of the procedure
- Other free identifiers in ⟨*s*⟩ are called *external references* ⟨z_1⟩ ... ⟨z_k⟩
- 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 ⟨*s*⟩

(proc {$ ⟨y_1⟩ ... ⟨y_n⟩ }
   ⟨*s*⟩
end ,
CE)

Procedure values (4)

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

(proc {$ ⟨y_1⟩ ... ⟨y_n⟩ }
   ⟨*s*⟩
end ,
CE)

Procedure introduction

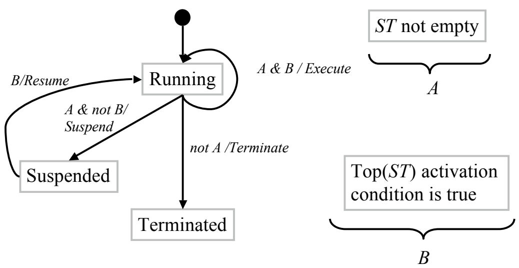
- The semantic statement is
  (⟨*x*⟩ = proc {$ ⟨y_1⟩ ... ⟨y_n⟩} ⟨*s*⟩ end, *E*)
- Create a contextual environment:
  $CE = E\,|_{\{\langle z_1\rangle \ldots \langle z_k\rangle\}}$ where ⟨z_1⟩ ... ⟨z_k⟩ are external references in ⟨*s*⟩.
- Create a new procedure value of the form:
  (proc {$ ⟨y_1⟩ ... ⟨y_n⟩} ⟨*s*⟩ end, *CE*) , refer to it by the variable x_P
- Bind the store variable *E*(⟨*x*⟩) 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 ⟨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 $\langle s_1 \rangle$ else $\langle s_2 \rangle$ 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 ($\langle s_1 \rangle$, E) on the stack
  - E(⟨x⟩) is false, push ($\langle s_2 \rangle$, E) on the stack
- If the activation condition (E(⟨x⟩) is determined) is false:
  - Suspend

Procedure application

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

Case statement

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

Execution examples

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

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


Execution examples (2)

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

```
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
```

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

Execution examples (3)

```
        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
          ⟨s⟩4 {Max 3 5 C}
        end
```

- After Max binding
  ([(⟨s⟩$_4$, {Max → m, C → c})],
  {m = (proc{$ X Y Z} ⟨s⟩$_3$ end , ∅) , c})
- After procedure call
  ([(⟨s⟩$_3$, {X → t_1, Y → t_2, Z → c})],
  {m = (proc{$ X Y Z} ⟨s⟩$_3$ end , ∅) , t_1=3, t_2=5, c})

Execution examples (4)

```
        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
          ⟨s⟩4 {Max 3 5 C}
        end
```

- After procedure call
  ([(⟨s⟩$_3$, {X → t_1, Y → t_2, Z → c})],
  {m = (proc{$ X Y Z} ⟨s⟩$_3$ end , ∅) , t_1=3, t_2=5, c})
- After T = (X>=Y)
  ([(⟨s⟩$_3$, {X → t_1, Y → t_2, Z → c, T → t})],
  {m = (proc{$ X Y Z} ⟨s⟩$_3$ end , ∅) , t_1=3, t_2=5, c, t=false})
- ([(Z=Y , {X → t_1, Y → t_2, Z → c, T → t})],
  {m = (proc{$ X Y Z} ⟨s⟩$_3$ end , ∅) , t_1=3, t_2=5, c, t=false})

Execution examples (5)

```
        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
          ⟨s⟩4 {Max 3 5 C}
        end
```

- ([(Z=Y , {X → t_1, Y → t_2, Z → c, T → t})],
  {m = (proc{$ X Y Z} ⟨s⟩$_3$ end , ∅) , t_1=3, t_2=5, c, t=false})
- ([ ],
  {m = (proc{$ X Y Z} ⟨s⟩$_3$ end , ∅) , t_1=3, t_2=5, c=5, t=false})

Exercises

50. Does dynamic binding require keeping an environment in a closure (procedure value)? Why or why not?
51. VRH Exercise 2.9.2 (page 107)
52. *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]}
```