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

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 $\sigma$ 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 $\sigma$, e.g. $\{X \rightarrow x_1, Y \rightarrow x_2\}$
- A semantic statement is a pair $\langle (s), E \rangle$ where $(s)$ is a statement
- $ST$ is a stack of semantic statements

Kernel language syntax

The following defines the syntax of a statement, $(s)$ denotes a statement:

$$(s) ::= \text{skip} \quad \text{empty statement}$$

$$\langle x \rangle = \langle y \rangle \quad \text{variable-variable binding}$$

$$\langle x \rangle = \langle v \rangle \quad \text{variable-value binding}$$

$$\langle s_1 \rangle \langle s_2 \rangle \quad \text{sequential composition}$$

$$\text{local} \langle x \rangle \text{in} \langle s \rangle \text{end} \quad \text{declaration}$$

$$\text{if} \langle x \rangle \text{then} \langle s_1 \rangle \text{else} \langle s_2 \rangle \text{end} \quad \text{conditional}$$

$$\{ \langle x \rangle \langle y_1 \rangle \ldots \langle y_n \rangle \} \quad \text{procedural application}$$

$$\text{case} \langle x \rangle \text{of} \langle \text{pattern} \rangle \text{then} \langle s_1 \rangle \text{else} \langle s_2 \rangle \text{end} \quad \text{pattern matching}$$

$$\langle v \rangle ::= \text{proc} \{ \langle y_1 \rangle \ldots \langle y_n \rangle \} \langle s_1 \rangle \text{end} \quad \text{value expression}$$

$$\langle \text{pattern} \rangle ::= \ldots \quad \text{...}$$

Computations (abstract machine)

- A computation is a sequence of execution states
  - 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 $(\langle (s), E \rangle, \emptyset)$
- $ST$ has a single semantic statement $(\langle (s), E \rangle)$
- The environment $E$ is empty, and the store $\sigma$ is empty
- $\ldots$ 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
**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 $(\text{local } X \text{ in } \langle s \rangle \text{ end}, E)$
- 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-variable equality**

- The semantic statement is $(\langle x \rangle = \langle y \rangle, E)$
- Bind $E(x)$ and $E(y)$ 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 \(E((x))\) 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
  \(\langle p\rangle = \{\langle y_1 \rangle \ldots \langle y_n \rangle\} \langle s\rangle end, CE\)
- \(\langle y_1 \rangle \ldots \langle y_n \rangle\) are the (formal) parameters of the procedure
- Other free identifiers of \(\langle s\rangle\) are called external references \(\langle z_1 \rangle \ldots \langle z_k \rangle\)
- 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 of \(\langle s\rangle\)

Procedure values (4)

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

Procedure introduction

- The semantic statement is \(\langle s\rangle = \text{proc } \{y_1 \ldots y_n\} \langle s\rangle end, E\)
- Environment is \(\langle s\rangle \rightarrow x_{s}\)
- Create a new procedure value of the form:
  \((\text{proc } \{y_1 \ldots y_n\} \langle s\rangle end, CE)\)
- Bind the store variable \(E((s))\) to \(x_{s}\)
- 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, i.e., \( x \) is bound to a number, record or a procedure value.

\[
\begin{align*}
  x & := \ldots \\
  | \text{ if } (x) \text{ then } (s_1) \text{ else } (s_2) & \text{ end } & \text{ conditional} \\
  | \{ (x) \} \ldots \{ y \} & \text{ procedural application} \\
  | \text{ case } (x) \text{ of } \{ \text{ (pattern) then } (s_1) \} & \text{ pattern matching} \\
  | \text{ else } (s_2) & \text{ end }
\end{align*}
\]

### Life cycle of a thread

- Execution examples

### Conditionals

- The semantic statement is \( (((x) \ (y)) \ldots (z)) \), \( E \).
- The activation condition \( E((x)) \) is determined) is true:
  - If \( E((x)) \) is not Boolean (false, true), 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 \( (((x) \ (y)) \ldots (z)) \), \( E \).
- 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 \( \text{proc} \{S (z_1) \ldots (z_n) \} \ (s) \), \( CE \), push \( (x) \), \( CE + (z_1) \rightarrow E((y_1)) \ldots (z_2) \rightarrow E((y_2)) \) on the stack.
- The activation condition \( E((x)) \) is determined) is false: suspend.

### Case statement

- The semantic statement is \( \text{case } (x) \text{ of } \{ (f_1) \ldots (f_n) \} \ldots (f_m) \ (s_1) \).
- The activation condition \( E((x)) \) is determined) is true:
  - The label of \( E((x)) \) is \( \emptyset \) and its arity is \( \{ f_1 \ldots (f_m) \} \): push \( \text{local } (x_1) \rightarrow (s_1), (x_2) \ldots (x_n) \rightarrow (s_n) \) in \( (s) \) end, \( E \) on the stack.
  - Otherwise push \( ((s_2), E) \) on the stack.
- The activation condition \( E((x)) \) is determined) is false: suspend.
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
      end
    end
  end
```

Use the operational semantics to execute the call

```
[AddList [1 2] [4 5]]
```