### Declarative Computation Model

**Kernel language semantics**

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

Carlos Varela  
RPI  
October 16, 2009

Adapted with permission from:  
Seif Haridi  
KTH  
UCL  
RPI

---

### 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

\[
(s) \ := \ \begin{align*}
\text{skip} & \quad \text{empty statement} \\
(x) = (y) & \quad \text{variable-variable binding} \\
(x) = v & \quad \text{variable-value binding} \\
(x_1)(x_2) & \quad \text{sequential composition} \\
\text{local}(x) \ \text{in} \ (x_1) & \quad \text{declaration} \\
\text{if} \ (x) \ \text{then} \ (x_1) \ \text{else} \ (x_2) & \quad \text{conditional} \\
\text{case} \ (x) \ \text{of} \ (\text{pattern}) & \quad \text{pattern matching} \\
\text{proc} \ (\{ y_1, \ldots, y_n \}) (x_k) & \quad \text{value expression} \\
\text{pattern} & \ := \ ... 
\end{align*}
\]

---

### 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 \( (\theta, E) \) where \( \theta \) 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 \( (ST, \sigma) \)
  - where \( ST \) is a stack of semantic statements and \( \sigma \) is a single assignment store
- A computation in a sequence of execution states
  \[
  \begin{align*}
  (ST_0, \sigma_0) \rightarrow (ST_1, \sigma_1) \rightarrow (ST_2, \sigma_2) \rightarrow ...
  \end{align*}
  \]

---

### Semantics

- To execute a program (i.e., a statement) \( \theta \) the initial execution state is \( (\{(\theta)\}, \emptyset) \)
- \( ST \) has a single semantic statement \( ((\theta), \emptyset) \)
- The environment \( E \) is empty, and the store \( \sigma \) 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 \((\text{skip}, E)\)
- Continue to next execution step

\[
\begin{array}{c}
\sigma \\
\text{ST}
\end{array}
\overset{\text{\(\text{skip}\)}}{\longrightarrow}
\begin{array}{c}
\sigma \\
\text{ST}
\end{array}
\]

**Sequential composition**

- The semantic statement is \((\langle x \rangle \langle y \rangle, E)\)
- Push \((\langle x \rangle, E)\) and then push \((\langle y \rangle, E)\) on \(\text{ST}\)
- Continue to next execution step

\[
\begin{array}{c}
\sigma \\
\text{ST}
\end{array}
\overset{\text{\((\langle x \rangle \langle y \rangle, E)\)}}{\longrightarrow}
\begin{array}{c}
\sigma \\
\text{ST}
\end{array}
\overset{\text{\((\langle x \rangle, E)\)}}{\longrightarrow}
\begin{array}{c}
\sigma \\
\text{ST}
\end{array}
\]

**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 \((\langle x \rangle, E')\) on \(\text{ST}\)
- Continue to next execution step

\[
\begin{array}{c}
\sigma \\
\text{ST}
\end{array}
\overset{\text{\((\text{local} (x) \text{ in } \langle s \rangle \text{ end}, E)\)}}{\longrightarrow}
\begin{array}{c}
\sigma \\
\text{ST}
\end{array}
\overset{\text{\((\langle x \rangle, E')\)}}{\longrightarrow}
\begin{array}{c}
\sigma \\
\text{ST}
\end{array}
\]

**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 \(E + \langle X \rangle \rightarrow \langle x \rangle\), i.e. \(E'\) is the same as \(E\) but the identifier \(\langle X \rangle\) is mapped to \(\langle x \rangle\).
- Push \((\langle x \rangle, E')\) on \(\text{ST}\)
- Continue to next execution step

\[
\begin{array}{c}
\sigma \\
\text{ST}
\end{array}
\overset{\text{\((\text{local} X \text{ in } \langle s \rangle \text{ end}, E)\)}}{\longrightarrow}
\begin{array}{c}
\sigma \\
\text{ST}
\end{array}
\overset{\text{\((\langle x \rangle, E')\)}}{\longrightarrow}
\begin{array}{c}
\sigma \\
\text{ST}
\end{array}
\]

**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 \((\langle x \rangle = \langle y \rangle, E)\)
- Where \(\langle y \rangle\) 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
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
end
```

Procedure values (3)

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

```
\[
\begin{align*}
\text{proc } (y_1, \ldots, y_n) \to (s) \text{ end, } CE
\end{align*}
\]
```

Procedure values (4)

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

```
\[
\begin{align*}
\text{proc } (y_1, \ldots, y_n) \to (s) \text{ end, } CE
\end{align*}
\]

Procedure introduction

- The semantic statement is
  \((\sigma) = \text{proc } (y_1, \ldots, y_n) (s) \text{ end, } E)\)
- Create a contextual environment:
  \(CE = E | [z_1, \ldots, z_k]\) where \((z_1, \ldots, z_k)\) are external references in \((s)\).
- Create a new procedure value of the form:
  \(\text{proc } (y_1, \ldots, y_n) (s) \text{ end, } CE\), refer to it by the variable \(x_p\)
- Bind the store variable \(E(x_2)\) to \(x_p\)
- Continue to next execution step

```
\[
\begin{align*}
\text{proc } (y_1, \ldots, y_n) \to (s) \text{ end, } CE
\end{align*}
\]
```

Suspendable statements

- The remaining statements require \((x)\) to be bound in order to execute
- The activation condition \((E(x))\) is \textit{determined}, is that \((x)\) be bound to a number, a record or a procedure value

```
\[
\begin{align*}
\text{conditional } & \text{ procedural application } & \text{pattern matching } \\
\text{if } (x) \text{ then } (s_1) \text{ else } (s_2) \text{ end } & \{ (x) (y_1, \ldots, y_n) \} & \text{case } (x) \text{ of } \text{pattern} \text{ then } (s_1) \text{ else } (s_2) \text{ end }
\end{align*}
\]
```
If the activation condition (E(\text{x})) is determined) is true:
- If E(\text{x}) is not Boolean (true, false), raise an error
- E(\text{x}) is true, push ((s_b), E) on the stack
- E(\text{x}) is false, push ((s_b), E) on the stack

If the activation condition (E(\text{x})) is determined) is false:
- Suspend

• The semantic statement is
  \[
  \text{if } (\text{x}) \text{ then } (s_b) \text{ else } (s_a) \text{ end, } E
  \]

• If the activation condition (E(\text{x})) is determined) is true:
  - If E(\text{x}) is not a procedure value, or it is a procedure
    with arity that is not equal to \text{n}, raise an error
  - If E(\text{x}) is a record, and the label of E(\text{x}) is (\text{l}) and its arity is
    \[(\ell_f) \ldots (\ell_a)\]
    push (x, CE + \{(x, y) \rightarrow E(\text{y})\} \ldots (\text{x}, y) \rightarrow E(\text{y})), on the stack
- If the activation condition (E(\text{x})) is determined) is false:
  - Suspend

Case statement

• The semantic statement is
  \[
  \text{case } (\text{a}) \text{ of } (\ell_f) \ldots (\ell_a) \text{ end, } E
  \]
  \[
  (s_b) \text{ end, } E
  \]

- If the activation condition (E(\text{x})) is determined) is true:
  - If E(\text{x}) is a record, and the label \ell is (\text{l}) and its arity is
    \[(\ell_f) \ldots (\ell_a)\]
    push (local (x), f, C) \rightarrow \text{true}, \ell, (\text{x}, y) \rightarrow E(\text{y}), \text{ on the stack}
  - Otherwise, push ((s_b), E) on the stack
- If the activation condition (E(\text{x})) is determined) is false:
  - Suspend

Execution examples (2)

- Initial state \(((\text{a}_1), (\text{a}_2)), (\text{a}_3))
- After local Max C in ...
  - \[(\text{a}_1), \{\text{Max} \rightarrow (m, C \rightarrow c)\}, \{m, c\}\]
- After Max binding
  - \[(\text{a}_1), \{\text{Max} \rightarrow (m, C \rightarrow c)\}, \{m = \text{proc}(3 X Y Z) \{b_2\} \text{ end, } (\text{a}_3), c\}\]
Execution examples (3)

\[
\begin{align*}
&\text{local Max C in} \\
&(\text{proc \{Max X Y Z\} (s3, m, C \rightarrow c, \{\} \}) \\
&\text{if } X \Rightarrow Y \text{ then } Z \Rightarrow X \text{ else } Z \Rightarrow Y \end{align*}
\]

• After Max binding
  \[
  ( [[s2], \{\text{Max } \rightarrow \text{m}, \text{C } \rightarrow \text{c}\}]), \{m = (\text{proc }\{X Y Z\} (s3, \emptyset), \{\} \}), \{\} \}
  \]

• After procedure call
  \[
  ( [[s2], \{X \rightarrow t1, Y \rightarrow t2, Z \rightarrow c\}]), \{m = (\text{proc }\{X Y Z\} (s3, \emptyset), \{\} \}), \{t1 = 3, t2 = 5, c\} \}
  \]

Execution examples (4)

\[
\begin{align*}
&\text{local Max C in} \\
&(\text{proc \{Max X Y Z\} (s3, m, C \rightarrow c, \{\} \}) \\
&\text{if } X \Rightarrow Y \text{ then } Z \Rightarrow X \text{ else } Z \Rightarrow Y \end{align*}
\]

• After procedure call
  \[
  ( [[s2], \{X \rightarrow t1, Y \rightarrow t2, Z \rightarrow c\}]), \{m = (\text{proc }\{X Y Z\} (s3, \emptyset), \{\} \}), \{t1 = 3, t2 = 5, c\} \}
  \]

• After T = (X\geq Y)
  \[
  ( [[s2], \{X \rightarrow t1, Y \rightarrow t2, Z \rightarrow c\}]), \{m = (\text{proc }\{X Y Z\} (s3, \emptyset), \{\} \}), \{t1 = 3, t2 = 5, c\} \}
  \]

Execution examples (5)

\[
\begin{align*}
&\text{local Max C in} \\
&(\text{proc \{Max X Y Z\} (s3, m, C \rightarrow c, \{\} \}) \\
&\text{if } X \Rightarrow Y \text{ then } Z \Rightarrow X \text{ else } Z \Rightarrow Y \end{align*}
\]

• After procedure call
  \[
  ( [[s2], \{X \rightarrow t1, Y \rightarrow t2, Z \rightarrow c\}]), \{m = (\text{proc }\{X Y Z\} (s3, \emptyset), \{\} \}), \{t1 = 3, t2 = 5, c\} \}
  \]

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:

\[
\begin{align*}
\text{fun } &\{\text{AddList } L1 \text{ L2}\} \\
&\text{case } L1 \text{ of} \\
&H1|T1 \text{ then} \\
&\text{case } L2 \text{ of} \\
&H2|T2 \text{ then} \\
&H1+H2|{\text{AddList } T1 \text{ T2}}
\end{align*}
\]

Use the operational semantics to execute the call

\[
\{\text{AddList } [1 \text{ 2}] [3 \text{ 4}]\} \]