Declarative Computation Model

Kernel language semantics revisited (VRH 2.4.5)
From kernel to practical language (VRH 2.6)
Exceptions (VRH 2.7)

Carlos Varela
RPI
October 15, 2007

Adapted with permission from:
Seif Haridi
KTH
Peter Van Roy
UCL

Sequential declarative computation model

- The kernel language semantics revisited.
  - Suspendable statements:
    - if,
    - case,
    - procedure application.
  - Procedure values:
    - Procedure introduction
    - Procedure application.

Conditional

- The semantic statement is
  \[
  \text{(if} \ (x) \ \text{then} \ (s_1) \ \text{else} \ (s_2) \ \text{end}, \ E)\]
- 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

Case statement

- The semantic statement is
  \[
  \text{(case} \ (x) \ \text{of} \ (l_1) \ (f_1): \ (x_1) \ \ldots \ (f_n): \ (x_n) \ \text{then} \ (s_1) \ \text{else} \ (s_2) \ \text{end}, \ E)\]
- If the activation condition \(E((x))\) is determined) is true:
  - If \(E((x))\) is a record, the label of \(E((x))\) is \(l_i\) and its arity is \(\langle f_1 \ldots f_n \rangle\):
    - push \((\text{local} \ (x_1) = (x). \ f_1 \ldots (x_n) = (x). \ f_n) \in (s_1)\end, E)\) on the stack
  - Otherwise, push \((s_2), E)\) on the stack
- If the activation condition \(E((x))\) is determined) is false:
  - Suspend

Procedure values

- Constructing a procedure value in the store is not simple because a procedure may have external references

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

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

```
proc {$} {Browse hello} end
Browse → x_0
```

Procedure values (2)
If the activation condition \((s)\) is determined an external reference \((s)\) is called external references \((s)\) are defined by the environment \(E\) where the procedure is declared lexical scoping

- The contextual environment of the procedure \(CE\) is when the procedure is called \(CE\) is used to construct the environment for execution of \(s\)

The semantic statement is

- \(\{s\}\) = \(\text{proc}\{\{s\}_1, \ldots, \{s\}_n\}\) \((s)\) \(E\)
- Create a contextual environment:
  \(CE = E|_{s/\ldots/s}\) where \(\{s\}_1\) \(\ldots\) \(\{s\}_n\) are external references in \(s\)
- Create a new procedure value of the form
  \(\text{proc}\{\{s\}_1, \ldots, \{s\}_n\}\) \((s)\) \(CE\), refer to it by the variable \(r_p\)
- Bind the store variable \(E(s)\) to \(r_p\)
- Continue to next execution step
Execution examples (4)

\begin{align*}
&\text{local Max C in} \\
&\begin{cases}
(\text{Max X Y Z}) \\
(\text{LB X Z}) \\
\text{if } X > Y \text{ then } Z \leftarrow X \text{ else } Z \leftarrow Y \end{cases} \\
&\text{end} \\
&\text{end}
\end{align*}

- The procedure value of LB is
- \((\text{proc} [\text{Max X Y Z}])\)
- \((\text{proc} [\text{LB X Z}])\)
- After procedure call
  \((\text{if} (X > Y) \text{ then } Z \leftarrow X \text{ else } Z \leftarrow Y)\)

C. Varela; Adapted w/permission from S. Haridi and P. Van Roy

Execution examples (5)

\begin{align*}
&\text{local Max C in} \\
&\begin{cases}
(\text{Max X Y Z}) \\
(\text{LB X Z}) \\
\text{if } X > Y \text{ then } Z \leftarrow X \text{ else } Z \leftarrow Y \end{cases} \\
&\text{end} \\
&\text{end}
\end{align*}

- The procedure value of LB is
- \((\text{proc} [\text{Max X Y Z}])\)
- \((\text{proc} [\text{LB X Z}])\)
- After procedure call
  \((\text{if} (X > Y) \text{ then } Z \leftarrow X \text{ else } Z \leftarrow Y)\)

C. Varela; Adapted w/permission from S. Haridi and P. Van Roy

Procedures with external references

\begin{align*}
&\text{local LB Y C in} \\
&\begin{cases}
Y = 5 \\
\text{proc} [\text{LB X Z}] \\
\text{if } X > Y \text{ then } Z \leftarrow X \text{ else } Z \leftarrow Y \end{cases} \\
&\text{end} \\
&\text{end}
\end{align*}

- The procedure value of LB is
- \((\text{proc} [\text{LB X Z}])\)
- The store is \(y = 5, \ldots\)
- \((\text{store} [\text{Y \rightarrow y}])\)
- \((\text{store} [\text{X \rightarrow x}, \text{LB \rightarrow lb}, \text{C \rightarrow c}, \text{T \rightarrow t}])\)
- \((\text{store} [\text{y = 5, lb = proc[LX Z} \text{end}, \text{Y \rightarrow y}])\)
- \((\text{store} [\text{y = 5, lb = proc[LX Z} \text{end}, \text{Y \rightarrow y}])\)
- \((\text{store} [\text{y = 5, lb = proc[LX Z} \text{end}, \text{Y \rightarrow y}])\)

C. Varela; Adapted w/permission from S. Haridi and P. Van Roy
Procedures with external references

\[ \begin{aligned} \text{local } & LB \ Y \ C \text{ in} \\ Y = 5 \ &= \text{proc} \ {LB \ X \ Z} \\ &\begin{cases} \text{if } X \geq Y \text{ then} \\ &Z = X \\ \text{else} \\ &Z = Y \\ \end{cases} \end{aligned} \]

\[ \text{end} \quad \text{end} \quad {LB \ 3 \ C} \]

\[ \begin{aligned} \text{\{Browse X\}} &\text{ in} \\ &\begin{cases} \text{if } X \geq Y \text{ then} \\ &Z = X \\ \text{else} \\ &Z = Y \\ \end{cases} \end{aligned} \]

\[ \text{end} \quad \text{end} \]

\[ \text{\{Browse X\}} \quad \text{\{Browse Y\}} \]

\[ \begin{aligned} \text{local } & A \ B \ C \ D \text{ in} \\ &T = \text{tree} (\text{key}\ A \ \text{left} B \ \text{right} C \ \text{value} D) \end{aligned} \]

From the kernel language to a practical language

• Interactive interface
  – the declare statement and the global environment
• Extend kernel syntax to give a full, practical syntax
  – nesting of partial values
  – implicit variable initialization
  – expressions
  – nesting the if and case statements
  – andthen and orelse operations
• Linguistic abstraction
  – Functions
  – Exceptions

The interactive interface (declare)

• The interactive interface is a program that has a single global environment
  
  \text{declare } X \ Y

• Augments (and overrides) the environment with new mappings for \(X\) and \(Y\)

\[ \{\text{Browse } X\} \]

• Inspects the store and shows partial values, and incremental changes

Syntactic extensions

• Nested partial values
  – person(name: “George” age:25)
    \begin{verbatim}
    local A B in “George” B=25 person(name: A age: B) end
    \end{verbatim}
• Implicit variable initialization
  – local (pattern) \(= \) (expression) in (statement) end
• Example:
  assume \(T\) has been defined, then
  \begin{verbatim}
  local tree(key:A left:B right:C value:D) = T in (statement) end
  \end{verbatim}
  is the same as:
  \begin{verbatim}
  local A B C D in
  T = tree(key:A left:B right:C value:D) <statement> end
  \end{verbatim}
Extracting fields in local statements

```plaintext
declare T :
T = tree(key: self.age:48 profession:professor):
local
tree(key:A ...) = T in
〈statement〉
end
```

Nested if and case statements

- Observe a pair notation is: T # 2, is the tuple "'(1 2)"
- case Xs # Ys of nil # Ys then (s1)
  - case Xs # Ys of X|Xr # (Y|Yr) then (s2) end
  - if X=<Y then (s3) else (s4) end
end

Expressions

- An expression is a sequence of operations that returns a value.
- A statement is a sequence of operations that does not return a value. Its effect is on the store, or outside of the system (e.g. read/write a file).

<table>
<thead>
<tr>
<th>Expression</th>
<th>Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>11*11</td>
<td>X=11*11</td>
</tr>
</tbody>
</table>

Functions as linguistic abstraction

- R = {F X1 ... Xn}
- (F X1 ... Xn R)

```
fun {F X1 ... Xn} ...
〈statement〉
〈expression〉
end

proc {F X1 ... Xn R} ...
〈statement〉
R = {expression}
end
```

Nesting in data structures

- Ys = {F X} {Map Xr F}
- Is unnested to:
  - local Y Yr in
    - Ys = Y|Yr
    - {F X Y}
    - {Map Xr F Yr}
end
- The unnesting of the calls occurs after the data structure

Functional nesting

- Nested notations that allows expressions as well as statements
- local R in
  - {F X1 ... Xn R}
  - {Q R ...}
end
- Is written as (equivalent to):
- {Q {F X1 ... Xn} ...} expression statement
Conditional expressions

\[
R = \begin{cases} 
\text{if } \langle \text{expr} \rangle \text{ then } \langle \text{expr} \rangle \text{ end} \\
\text{else } \langle \text{expr} \rangle \text{ end}
\end{cases}
\]

Example

\[
\text{fun} \{ \text{Max X Y} \}
\begin{cases} 
\text{if } X \geq Y \text{ then } X \\
\text{else } Y \end{cases}
\text{end}
\]

\[
\text{proc} \{ \text{Max X Y R} \}
\begin{cases} 
\text{if } X \geq Y \text{ then } R = X \\
\text{else } R = Y \end{cases}
\text{end}
\]

andthen and orelse

\[
\langle \text{expr} \rangle \text{ andthen } \langle \text{expr} \rangle
\]

\[
\langle \text{expr} \rangle \text{ orelse } \langle \text{expr} \rangle
\]

Function calls

\[
\text{local } R1 R2 \text{ in } R1 = \{ \text{F2 X} \}
R2 = \{ \text{F3 Y} \}
\text{end}
\]

The arguments of a function are evaluated first from left to right.

A complete example

\[
\text{fun} \{ \text{Map Xs F} \}
\begin{cases} 
\text{case Xs of } \text{nil then nil } \\
\ldots \text{X|Xr then } \langle F X \rangle | (\text{Map Xr F} ) \end{cases}
\text{end}
\]

\[
\text{proc} \{ \text{Map Xs F Ys} \}
\begin{cases} 
\text{case Xs of } \text{nil then Ys = nil } \\
\ldots \text{X|Xr then } \{ Y | Yr \} \text{ in } \langle F X Y \rangle | (\text{Map Xr F Yr} ) \end{cases}
\text{end}
\]

Exceptions

- How to handle exceptional situations in the program?
- Examples:
  - divide by 0
  - opening a nonexistent file
- Some errors are programming errors
- Some errors are imposed by the external environment
- Exception handling statements allow programs to handle and recover from errors
Exceptions

- The error confinement principle:
  - Define your program as a structured layers of components
  - Errors are visible only internally and a recovery procedure corrects
    the errors: either errors are not visible at the component boundary
    or are reported (nicely) to a higher level
- In one operation, exit from arbitrary depth of nested contexts
  - Essential for program structuring; else programs get complicated
    (use boolean variables everywhere, etc.)

Basic concepts

- A program that encounters an error (exception) should
  transfer execution to another part, the exception handler
  and give it a (partial) value that describes the error
- `try` `catch` `then` `end`
- `raise` `end`
- Introduce an exception marker on the semantic stack
- The execution is equivalent to `if` it executes without
  raising an error
- Otherwise, `else` is aborted and the stack is popped up to the
  marker, the error value is transferred through `catch`, and
  `end` is executed

Exceptions (Example)

```plaintext
fun {Eval E}  
  if {IsNumber E} then E  
  else  
    case E of  
      plus(X Y) then {Eval X}+{Eval Y}  
      times(X Y) then {Eval X}*{Eval Y}  
      else raise illFormedExpression(E) end  
    end  
  end  
end
```

Try semantics

- The semantic statement is
  - `try` `catch` `then` `end`, `E`
- Push the semantic statement `catch` `then` `end`, `E` on `ST`
- `end`
- Continue to next execution step

Raise semantics

- The semantic statement is
  - `raise` `end`, `E`
- Pop elements off `ST` looking for a `catch` statement:
  - If a `catch` statement is found, pop it from the stack
  - If the stack is emptied and no `catch` is found, then stop execution
    with the error message “Uncaught exception”
- Let `catch` `then` `end`, `E` be the `catch` statement that is found
- Push `catch`, `E` on `ST`
- Continue to next execution step
Catch semantics

- The semantic statement is
  \( \text{catch } (x) \text{ then } (s) \text{ end, } E \) 
- Continue to next execution step (like skip)

Full exception syntax

- Exception statements (expressions) with multiple patterns and finally clause
- Example:
  ```
  FH = \{\text{OpenFile "xxxx"} \}
  \{\text{try}
  \{\text{ProcessFile FH} \}
  \text{catch } X \text{ then}
  \{\text{System.showInfo "***** Exception when processing ***** # } X \}
  \text{finally} \{\text{CloseFile FH} \}
  \text{end}
  ```

Exercises

53. VRH Exercise 2.9.3 (page 107).
54. VRH Exercise 2.9.7 (page 108) – translate example to kernel language and execute using operational semantics.
55. 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.
56. *VRH Exercise 2.9.12 (page 110).
57. *Change the semantics of the case statement, so that patterns can contain variable labels and variable feature names.
58. *Restrict the kernel language to make it strictly functional (i.e., without dataflow variables)
   - Language similar to Scheme (dynamically typed functional language)
   - This is done by disallowing variable declaration (without initialization) and disallowing procedural syntax
     - Only use implicit variable initialization
     - Only use functions