Sequential declarative computation model

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

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 2, 2006

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

Conditional

- The semantic statement is
  (if (x) then (s1) else (s2) end, E)
- If the activation condition (E(x)) is determined) is true:
  - If E(x) is not Boolean (true, false), raise an error
  - If E(x) is true, push ((s1), E) on the stack
  - If E(x) is false, push ((s2), E) on the stack
- If the activation condition (E(x)) is determined) is false:
  - Suspend

Case statement

- The semantic statement is
  (case (x) of (l1) (f1): (x1) ... (ln) (fn): (xn))
    then (s1) end
    else (s2) end, E
- If the activation condition (E(x)) is determined) is true:
  - If E(x) is a record, the label of E(x) is (l1) and its arity is (f1) ... (fn):
    push (local (x1) = (x). (f1) ... (xn) = (x). (ln) in (s1)) end, E
    on the stack
  - Otherwise, push ((s2), 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

```prolog
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
end
```

```
proc [$] { [Browse hello] end } P
proc [$] { Q} end 
Q → x1
Browse → x1
P → x2
```
The semantic statement is
\( \text{if} (\text{E}(x)) \) is determined (true):  
- If \( \text{E}(x) \) is not a procedure value, or it is a procedure
  with arity that is not equal to \( n \), raise an error
- If \( \text{E}(x) \) is \( (\text{proc} \{ \{ y \} \} \{ z \}) \) \( \text{CE} \),
  push \( (x, CE + \{ y \} \rightarrow \text{E}(y) \} \}
  on the stack
- If the activation condition \( \text{E}(x) \) is determined (false):
  - Suspend

Create a contextual environment:
CE = E[\{x/y\} – \{z/k\}]

where \( \{x/y\} \) and \( \{z/k\} \) are external references to \( x \) and \( z \).

Create a new procedure value of the form:
(\text{proc} \{ \{ y \} \} \{ z \}) \( \text{CE} \), refer to it by the variable \( s \).

Bind the store variable \( \text{E}(x) \) to \( s \).

Continue to next execution step
Execution examples (4)

\[\begin{align*}
\text{local Max C in} & \quad \text{proc} \ [\text{Max} \ Y \ Z] \\
\begin{cases}
(\delta_0) & \{Y = 5\} \\
(\delta_1) & \{\text{if } X \geq Y \text{ then } Z = X \text{ else } Z = Y \} \\
(\delta_2) & \{\text{LB } 3 \ C\}
\end{cases}
\end{align*}\]

- After procedure call
  \(\{y\} (X \rightarrow \text{if } t_2, Y \rightarrow t_2, Z \rightarrow \text{c})\)
  \(\{m = \text{proc} \{\text{Max} \ Y \ Z\} \ (\delta_1) \text{ end} , \ (\delta_2) , t_2=3, t_2=5, c\}\)
- After \(T = (X < Y)\)
  \(\{y\} (X \rightarrow \text{if } t_2, Y \rightarrow t_2, Z \rightarrow c, T \rightarrow \text{t})\)
  \(\{m = \text{proc} \{\text{Max} \ Y \ Z\} \ (\delta_1) \text{ end} , \ (\delta_2) , t_2=3, t_2=5, c, t\text{=false}\}\)
- \((\{Z=X, (X \rightarrow \text{if } t_2, Y \rightarrow t_2, Z \rightarrow c, T \rightarrow \text{t})\})
  \(\{m = \text{proc} \{\text{Max} \ Y \ Z\} \ (\delta_1) \text{ end} , \ (\delta_2) , t_2=3, t_2=5, c, t\text{=false}\}\)

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

---

Execution examples (5)

\[\begin{align*}
\text{local Max C in} & \quad \text{proc} \ [\text{Max} \ Y \ Z] \\
\begin{cases}
(\delta_0) & \{Y = 5\} \\
(\delta_1) & \{\text{if } X \geq Y \text{ then } Z = X \text{ else } Z = Y \} \\
(\delta_2) & \{\text{LB } 3 \ C\}
\end{cases}
\end{align*}\]

- \((\{Z=X, (X \rightarrow \text{if } t_2, Y \rightarrow t_2, Z \rightarrow c, T \rightarrow \text{t})\})
  \(\{m = \text{proc} \{\text{Max} \ Y \ Z\} \ (\delta_1) \text{ end} , \ (\delta_2) , t_2=3, t_2=5, c, t\text{=false}\}\)
- \((\{\} \{m = \text{proc} \{\text{Max} \ Y \ Z\} \ (\delta_1) \text{ end} , \ (\delta_2) , t_2=3, t_2=5, c, t\text{=false}\}\)

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

---

Procedures with external references

\[\begin{align*}
\text{local LB Y C in} & \quad \text{proc} \ [\text{LB} \ X \ Z] \\
\begin{cases}
(\delta_0) & \{Y = 5\} \\
(\delta_1) & \{\text{if } X \geq Y \text{ then } Z = X \text{ else } Z = Y \} \\
(\delta_2) & \{\text{LB } 3 \ C\}
\end{cases}
\end{align*}\]

- The procedure value of LB is
  \(\{\text{proc} \{\text{Max} \ Y \ Z\} \ (\delta_2) \text{ end} , \ (Y \rightarrow y)\}\)
- The store is \(\{y = 5, \ldots\}\)
- STACK: \([\text{LB } T \ C], \{Y \rightarrow y, \text{LB } \rightarrow \text{lb}, C \rightarrow c, T \rightarrow t\}\]
- STORE: \([y = 5, \text{lb} = \{\text{proc} \{\text{Max} \ Y \ Z\} \ (\delta_2) \text{ end} , \ (Y \rightarrow y)\}, t = 3, c]\)

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

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

From the kernel language to a practical language

- Interative interface
  - the \texttt{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
  - and then and or else operations
- Linguistic abstraction
  - Functions
  - Exceptions

The interactive interface (\texttt{declare})

- The interactive interface is a program that has a single global environment

\texttt{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) local A B in A: "George" B: 25 person(name: A age: B) end

- Implicit variable initialization
  - local (pattern) = (expression) in (statement) end

- Example:

  assume \(T\) has been defined, then
  
  local tree(key: A left: B right: C value: D) = T in (statement) end
  
  is the same as:
  
  local A B C D in
  
  \(T = \text{tree(key: A left: B right: C value: D) <statement> end}\)
Extracting fields in local statement

```verbatim
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 i.e. T # 2, is the tuple ‘1,2’
- Is translated into:
  ```verbatim
case Xs # Ys
of
  nil # Ys then (s1)
  Xs # nil then (s2)
  else (s3)
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)
- \( 11 \times 11 \) \( X=11 \times 11 \)

Functions as linguistic abstraction

- \( \{F X1 \ldots Xn R\} \)
- \( R = \{F X1 \ldots Xn\} \)

Nesting in data structures

- \( Ys = \{F X\}|\{Map Xr F\} \)
- Is unnested to:
  ```verbatim
  local Yr
  Ys = Yr|Y
  \{F X Y\}
  \{Map Xr H Y\}
end
```
- The unnesting of the calls occurs after the data structure

Functional nesting

- Nested notations that allows expressions as well as statements
- Is written as (equivalent to):
  ```verbatim
  proc \{F X1 \ldots Xn R\}
  (statement)
  (expression)
end
```

- Local R in:
  ```verbatim
  \{F X1 \ldots Xn R\}
  \{Q R\ ...
end
```
Conditional expressions

\[
R = \begin{cases} 
(\text{expr})_2 & \text{if } (\text{expr})_1 \\
(\text{expr})_3 & \text{else}
\end{cases}
\]

Example

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

\[
\text{proc} \{\text{Max X Y R}\} \\
\quad R = \begin{cases} 
(\text{expr})_2 & \text{if } X \geq Y \\
(\text{expr})_3 & \text{else}
\end{cases}
\]

andthen and orelse

\[
(\text{expr})_1 \text{andthen } (\text{expr})_2 
\]

\[
(\text{expr})_1 \text{orelse } (\text{expr})_2 
\]

Function calls

\[
\text{Observe} \\
[F1 \{F2 X\} \{F3 Y\}]
\]

A complete example

\[
\text{fun} \{\text{Map Xs F}\} \\
\quad \text{case Xs of} \\
\quad \text{nil then nil} \\
\quad \text{X|Xr then } (\text{F} X)(\text{Map Xr F})
\]

Exception handling

- 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 〈s〉1 catch 〈x〉 then 〈s〉2 end
- raise 〈x〉 end
- Introduce an exception marker on the semantic stack
- The execution is equivalent to 〈s〉1 if it executes without
  raising an error
- Otherwise, 〈s〉1 is aborted and the stack is popped up to the
  marker, the error value is transferred through 〈x〉, and
  〈s〉2 is executed

Exceptions (Example)

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

Exceptions (Example)

try {Browse {Eval plus(5 6) }} {Browse {Eval plus(times(5 5) 6) }}
{Browse {Eval plus(minus(5 5) 6) }}
catch illFormedExpression(E) then
  {System.showInfo **** illegal expression **** # E}
end

Try semantics

- The semantic statement is
  try 〈x〉1 catch 〈y〉 then 〈x〉2 end, E
- Push the semantic statement (catch 〈y〉 then 〈x〉2 end, E) on ST
- Push 〈(s)1 , E〉 on ST
- Continue to next execution step

Raise semantics

- The semantic statement is
  (raise 〈x〉 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 〈y〉 then 〈x〉 end, E) be the catch statement that is found
- Push 〈(s) , E, E+〈y→E(〈x〉)〉〉 on ST
- Continue to next execution step
Catch semantics

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

Full exception syntax

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

Exercises

34. VRH Exercise 2.9.3 (page 107).
35. VRH Exercise 2.9.7 (page 108) – translate example to kernel language and execute using operational semantics.
36. 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.
38. *Change the semantics of the \text{case} statement, so that patterns can contain variable labels and variable feature names.

Exercises

39. *Restrict the kernel language to make it strictly functional (i.e., without dataflow variables)
   - Language similar to \text{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