Declarative Computation Model

From kernel to practical language (CTM 2.6)  
Exceptions (CTM 2.7)  

Carlos Varela  
RPI  
March 30, 2015

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

\texttt{declare} X Y

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

\{Browse X\}

• Inspects the store and shows partial values, and incremental changes
The interactive interface (declare)

Environment
- Browse
- R
- X
- Y

procedure value

value

a

b

Store
declare $X \ Y$

```
procedure value
  value
  a
  b
unbound $x_i$
unbound $x_{i+1}$
```

Environment

Browse

R

X

Y

Store
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 = tree(key:A left:B right:C value:D) <statement>
end
```
Extracting fields in local statement

```
declare T :
  T = tree(key:seif age:48 profession:professor)
 :
local
  tree(key:A ...) = T
in
<statement>
end
```
Nested if and case statements

- Observe a pair notation is: \(1 # 2\), is the tuple ‘#’ (1 2)

\begin{verbatim}
case Xs # Ys
  of  nil # Ys then \langle s \rangle_1
  │ Xs # nil then \langle s \rangle_2
  │ (X|Xr) # (Y|Yr) andthen X=<Y then \langle s \rangle_3
  else \langle s \rangle_4 end
end
\end{verbatim}

- Is translated into (assuming X,Xr,Y,Yr not free in \(\langle s \rangle_4\))

\begin{verbatim}
case Xs of nil then \langle s \rangle_1
  else
    case Ys of nil then \langle s \rangle_2
            else
              case Xs of X|Xr then
                case Ys of Y|Yr then
                  if X=<Y then \langle s \rangle_3 else \langle s \rangle_4 end
                else \langle s \rangle_4 end
              else \langle s \rangle_4 end
            else \langle s \rangle_4 end
  end
end
\end{verbatim}

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

• $11 \times 11$  
  expression

• $X = 11 \times 11$  
  statement
Functions as linguistic abstraction

- $R = \{F \ X_1 \ldots \ X_n\}$
- $\{F \ X_1 \ldots \ X_n \ R\}$

```plaintext
fun \{F \ X_1 \ldots \ X_n\} 
  \langle statement \rangle
  \langle expression \rangle
end

F = proc \{$ \ X_1 \ldots \ X_n \ R$\}
  \langle statement \rangle
R = \langle expression \rangle
end
```

C. Varela; Adapted w/permission from S. Haridi and P. Van Roy
Nesting in data structures

• \( Y_s = \{F \times\} | \{\text{Map} \times r \ F\} \)

• Is unnested to:

  \[
  \text{local} \ Y \ Y_r \ \text{in} \\
  Y_s = Y | Y_r \\
  \{F \times Y\} \\
  \{\text{Map} \times r \ F \ Y_r\}
  \]

• The unnesting of the calls occurs after the data structure
Functional nesting

• Nested notations that allows expressions as well as statements
  
  \[ \text{local } R \text{ in} \]
  \[ \{ F \ X_1 \ldots \ X_n \ R \} \]
  \[ \{ Q \ R \ldots \} \]
  \[ \text{end} \]
  
  Is written as (equivalent to):
  
  \[ \{ Q \ \{ F \ X_1 \ldots \ X_n \} \ldots \} \]

expression

statement
Conditional expressions

\[
R = \text{if } \langle \text{expr} \rangle_1 \text{ then } \\
\quad \langle \text{expr} \rangle_2 \\
\text{else } \langle \text{expr} \rangle_3 \text{ end}
\]

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

Max = \text{proc } \{X Y R\} \\
\quad R = (\text{if } X \geq Y \text{ then } X \\
\quad \text{else } Y \text{ end} ) \\
end
Example

fun {Max X Y}
   if X>=Y then X
   else Y end
end

Max = proc {X Y R}
   R = ( if X>=Y then X
        else Y end )
end

Max = proc {X Y R}
   if X>=Y then R = X
   else R = Y end
end
andthen and orelse

if \langle expr \rangle_1 \text{ then } \langle expr \rangle_2 \\
\text{ else false end}

\langle expr \rangle_1 \text{ orelse } \langle expr \rangle_2 \\
\text{ if } \langle expr \rangle_1 \text{ then } True \\
\text{ else } \langle expr \rangle_2 \text{ end}
Function calls

Observe

\{F1 \{F2 X\} \{F3 Y\}\}

local R1 R2 in
\{F2 X R1\}
\{F3 Y R2\}
\{F1 R1 R2\}
end

The arguments of a function are evaluated first from left to right
A complete example

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

proc \{\text{Map } Xs \ F \ Ys\}
   \text{case } Xs
      \text{of } \text{nil then } Ys = \text{nil}
      \text{[] } X|Xr \text{ then } Y \ Yr \text{ in}
      \quad Ys = Y|Yr
      \quad \{F X \ Y\}
      \quad \{\text{Map } Xr \ F \ Yr\}
   \text{end}
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

  • \texttt{try} \langle s \rangle_1 \texttt{catch} \langle x \rangle \texttt{then} \langle s \rangle_2 \texttt{end}
  
  • \texttt{raise} \langle x \rangle \texttt{end}

• Introduce an exception marker on the semantic stack

• The execution is equivalent to \langle s \rangle_1 if it executes without raising an error

• Otherwise, \langle s \rangle_1 is aborted and the stack is popped up to the marker, the error value is transferred through \langle x \rangle, and \langle s \rangle_2 is executed
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 expresion ****" # E}
end
Try semantics

- The semantic statement is 
  \[(\text{try } \langle s \rangle_1 \text{ catch } \langle y \rangle \text{ then } \langle s \rangle_2 \text{ end, } E)\]
- Push the semantic statement \((\text{catch } \langle y \rangle \text{ then } \langle s \rangle_2 \text{ end, } E)\) on \(ST\)
- Push \((\langle s \rangle_1 , E)\) on \(ST\)
- Continue to next execution step
Raise semantics

• The semantic statement is
  \((\text{raise } \langle x \rangle \text{ end, } E)\)

• Pop elements off \(ST\) looking for a \texttt{catch} statement:
  – If a \texttt{catch} statement is found, pop it from the stack
  – If the stack is emptied and no \texttt{catch} is found, then stop execution
    with the error message ”Uncaught exception”

• Let \((\text{catch } \langle y \rangle \text{ then } \langle s \rangle \text{ end, } E_c)\) be the \texttt{catch}
  statement that is found

• Push \((\langle s \rangle, E_c + \{<y> \rightarrow E(<x>)\})\) on \(ST\)

• Continue to next execution step
Catch semantics

• The semantic statement is

\[(\text{catch } \langle x \rangle \text{ then } \langle s \rangle \text{ end}, E)\]

• Continue to next execution step (like \textit{skip})
Exception statements (expressions) with multiple patterns and **finally** clause

Example:

: FH = {OpenFile "xxxxx"}
: try
   {ProcessFile FH}
   catch X then
   {System.showInfo "***** Exception when processing *****" # X}
finally {CloseFile FH} end
finally syntax

\[
\text{try } \langle s \rangle_1 \text{ finally } \langle s \rangle_2 \text{ end}
\]

is converted to:

\[
\begin{align*}
\text{try } & \langle s \rangle_1 \\
\text{catch } & X \text{ then} \\
\langle s \rangle_2 & \\
\text{raise } & X \text{ end} \\
\text{end} \\
\langle s \rangle_2
\end{align*}
\]
52. CTM Exercise 2.9.12 (page 110).

53. Change the semantics of the case statement, so that patterns can contain variable labels and variable feature names.

54. 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