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

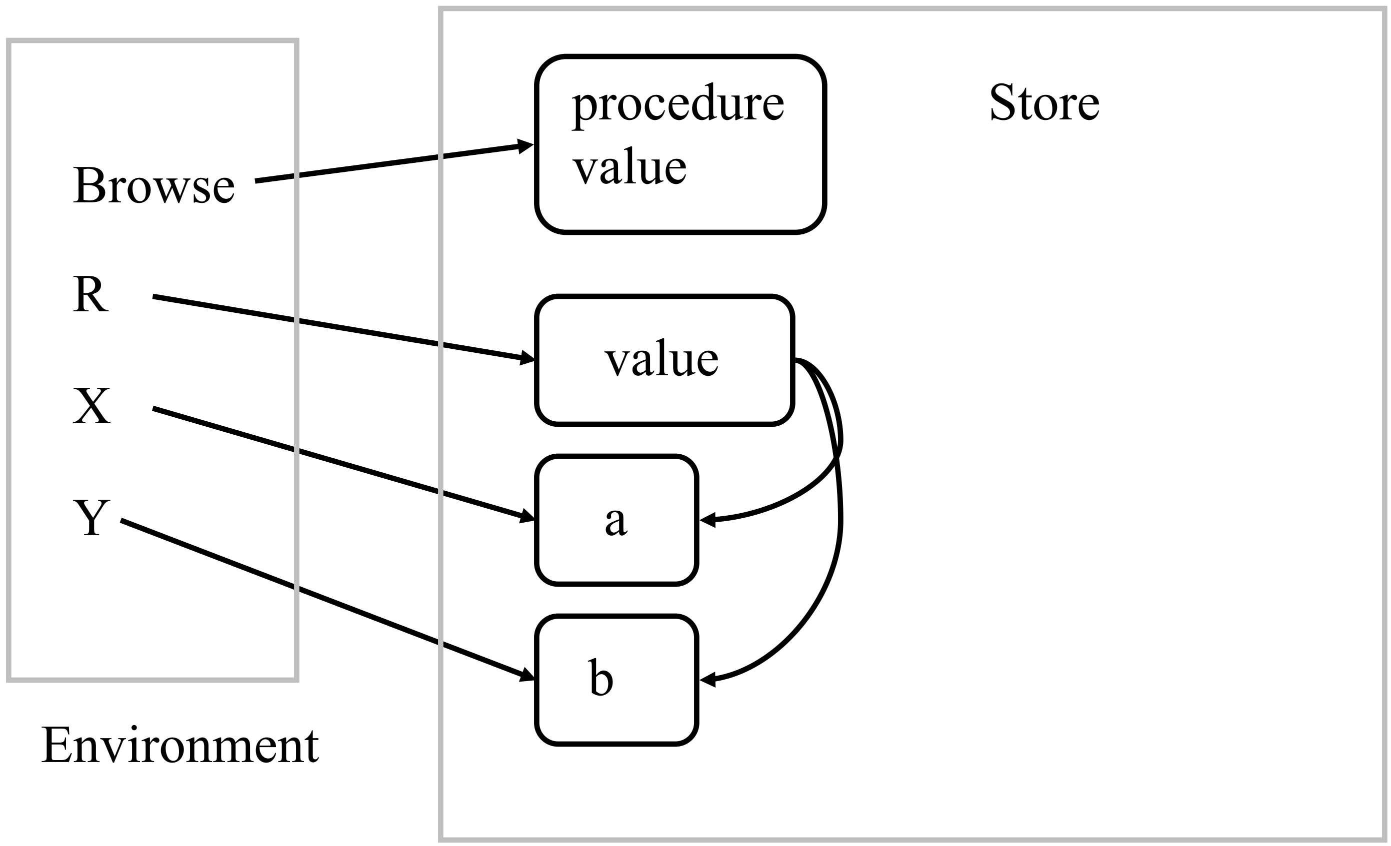
```
declare X Y
```

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

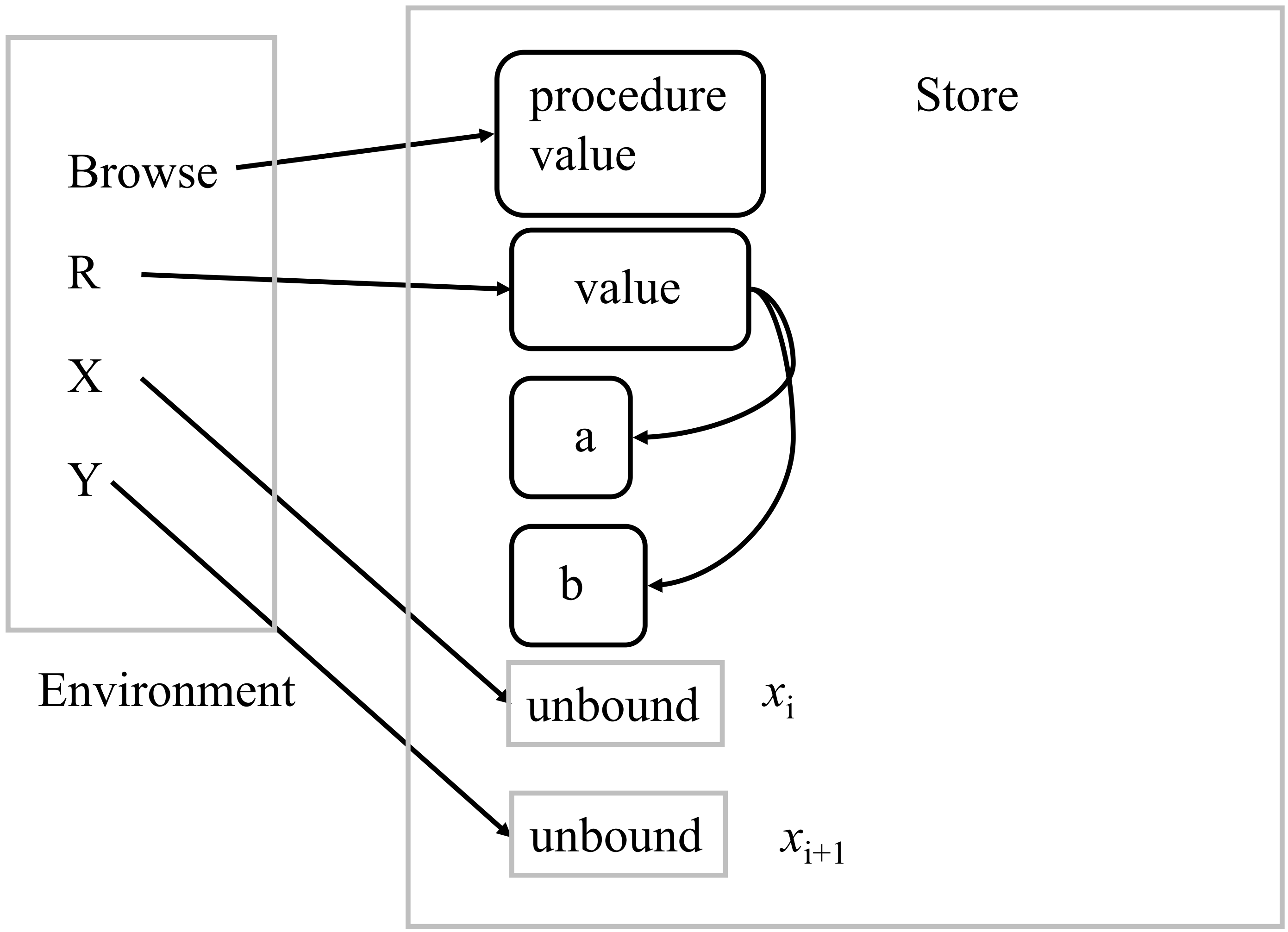
```
{Browse X}
```

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

The interactive interface (declare)



declare X Y



Browse
R
X
Y
Environment
procedure value
value
a
b
unbound
x_i
unbound
x_{i+1}
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)

```
case Xs # Ys
of  nil # Ys then ⟨s⟩1
[]  Xs # nil then ⟨s⟩2
[]  (X|Xr) # (Y|Yr) andthen X=<Y then ⟨s⟩3
else ⟨s⟩4 end
```

- Is translated into (assuming X,Xr,Y,Yr not free in ⟨s⟩4)

```
case Xs of nil then ⟨s⟩1
else
    case Ys of nil then ⟨s⟩2
    else
        case Xs of X|Xr then
            case Ys of Y|Yr then
                if  X=<Y then ⟨s⟩3 else ⟨s⟩4 end
            else ⟨s⟩4 end
        else ⟨s⟩4 end
    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)
- $\underbrace{11*11}_{\text{expression}}$ $\qquad\qquad$ $\underbrace{X=11*11}_{\text{statement}}$

Functions as linguistic abstraction

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

```
fun {F X1 ... Xn}
    ⟨statement⟩
    ⟨expression⟩
end
```

⟨statement⟩

```
F = proc {$ X1 ... Xn R}
    ⟨statement⟩
    R = ⟨expression⟩
  end
```

⟨statement⟩

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):
- $\underbrace{\texttt{\{Q }\underbrace{\texttt{\{F X1 ... Xn\}}}_{\text{expression}}\texttt{ ...\}}}_{\text{statement}}$

Conditional expressions

$\underbrace{\text{R = if } \langle\text{expr}\rangle_1 \text{ then } \langle\text{expr}\rangle_2 \text{ else } \langle\text{expr}\rangle_3 \text{ end}}_{\langle\text{expression}\rangle}$

$\Rightarrow$

$\underbrace{\text{if } \langle\text{expr}\rangle_1 \text{ then R = } \langle\text{expr}\rangle_2 \text{ else R = } \langle\text{expr}\rangle_3 \text{ end}}_{\langle\text{statement}\rangle}$

```
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
```

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

$\langle \text{expr} \rangle_1$ andthen $\langle \text{expr} \rangle_2$ ⇒

```
if ⟨expr⟩1 then
    ⟨expr⟩2
else false end
```

$\langle \text{expr} \rangle_1$ orelse $\langle \text{expr} \rangle_2$ ⇒

```
if ⟨expr⟩1 then
    true
else ⟨expr⟩2 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 {Map Xs F}
    case Xs
    of nil then nil
    []  X|Xr then {F X}|{Map Xr F}
    end
end
```

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

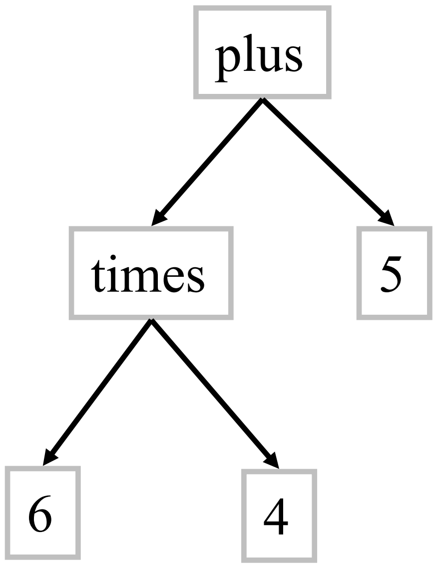
- The error confinement principle:
  - Define your program as a structured layers of components
  - Errors are visible only internally and a recovery procedure corrects the errors: either errors are not visible at the component boundary or are reported (nicely) to a higher level
- In one operation, exit 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
- try $\langle s \rangle_1$ catch $\langle x \rangle$ then $\langle s \rangle_2$ end
- raise $\langle x \rangle$ 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

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
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

  (try $\langle s\rangle_1$ catch $\langle \mathrm{y}\rangle$ then $\langle s\rangle_2$ end, E)
- Push the semantic statement (catch $\langle \mathrm{y}\rangle$ then $\langle s\rangle_2$ end, E) on ST
- Push $(\langle s\rangle_1, E)$ on ST
- Continue to next execution step

Raise semantics

- The semantic statement is
  (raise $\langle x \rangle$ 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 $\langle y \rangle$ then $\langle s \rangle$ end, E_c) be the 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
  (catch $\langle \mathrm{x} \rangle$ then $\langle s \rangle$ end, E)
- Continue to next execution step (like skip)

Full exception syntax

- 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

try $\langle s \rangle_1$ finally $\langle s \rangle_2$ end

is converted to:

try $\langle s \rangle_1$
catch X then
   $\langle s \rangle_2$
   raise X end
end
$\langle s \rangle_2$

Exercises

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