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

March 21, 2011

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
  (if ⟨x⟩ then ⟨s_1⟩ else ⟨s_2⟩ 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

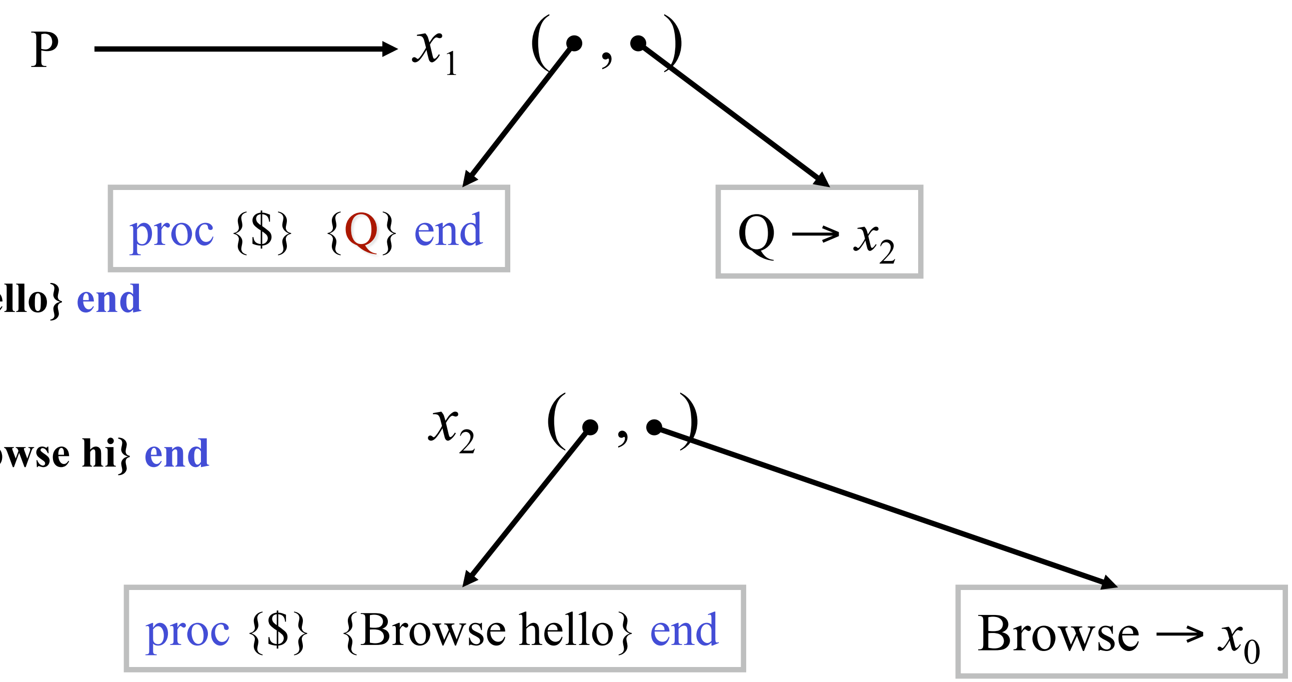
- The semantic statement is
  (case ⟨x⟩ of ⟨l⟩ (⟨f_1⟩ : ⟨x_1⟩ … ⟨f_n⟩ : ⟨x_n⟩)
  then ⟨s_1⟩
  else ⟨s_2⟩ end , E)
- If the activation condition (E(⟨x⟩) is determined) is true:
  - If E(⟨x⟩) is a record, the label of E(⟨x⟩) is ⟨l⟩ and its arity is [⟨f_1⟩ … ⟨f_n⟩]:
    push (local ⟨x_1⟩ = ⟨x⟩. ⟨f_1⟩ … ⟨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

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

Procedure values (3)

- The semantic statement is

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

$$(\text{proc } \{\$ \langle y_1 \rangle \ldots \langle y_n \rangle \} \; \langle s \rangle \; \text{end}, \; CE)$$

Procedure introduction

- The semantic statement is

  $(\langle x \rangle = \texttt{proc}\ \{\$\ \langle y_1 \rangle \ldots \langle y_n \rangle\}\ \langle s \rangle\ \texttt{end},\ E)$

- Create a contextual environment:

  $CE = E\,|_{\{\langle z_1 \rangle \ldots \langle z_k \rangle\}}$ where $\langle z_1 \rangle \ldots \langle z_k \rangle$ are external references in $\langle s \rangle$.

- Create a new procedure value of the form: $(\texttt{proc}\ \{\$\ \langle y_1 \rangle \ldots \langle y_n \rangle\}\ \langle s \rangle\ \texttt{end},\ CE)$, refer to it by the variable x_P
- Bind the store variable $E(\langle x \rangle)$ to x_P
- Continue to next execution step

Procedure application

- The semantic statement is

  $(\{ \langle x \rangle \langle y_1 \rangle \ldots \langle y_n \rangle \}, E)$
- If the activation condition ($E(\langle x \rangle)$ is determined) is true:
  - If $E(\langle x \rangle)$ is not a procedure value, or it is a procedure with arity that is not equal to n, raise an error
  - If $E(\langle x \rangle)$ is (proc {$ $\langle z_1 \rangle \ldots \langle z_n \rangle$} $\langle s \rangle$ end, CE) ,

    push

    $(\langle s \rangle , CE + \{\langle z_1 \rangle \rightarrow E(\langle y_1 \rangle) \ldots \langle z_n \rangle \rightarrow E(\langle y_n \rangle)\})$

    on the stack
- If the activation condition ($E(\langle x \rangle)$ is determined) is false:
  - Suspend

Execution examples

$\langle s \rangle_1$ { $\langle s \rangle_2$ { $\langle s \rangle_3$ }

```
local Max C in
   proc {Max X Y Z}
      if X >= Y then Z=X else Z=Y end
   end
   {Max 3 5 C}
end
```

- $\langle s \rangle_1$: the whole `local Max C in … end` statement
- $\langle s \rangle_2$: `proc {Max X Y Z} … end`
- $\langle s \rangle_3$: `if X >= Y then Z=X else Z=Y end`

Execution examples (2)

$\langle s \rangle_1$ { $\langle s \rangle_2$ { $\langle s \rangle_3$ $\langle s \rangle_4$ } }

```
local Max C in
   proc {Max X Y Z}
  ⟨s⟩3   if X >= Y then Z=X else Z=Y end
   end
 ⟨s⟩4 {Max 3 5 C}
end
```

- Initial state $([(\langle s \rangle_1, \varnothing)], \varnothing)$
- After local Max C in …
  $([(\langle s \rangle_2,\ \{\text{Max} \rightarrow m, \text{C} \rightarrow c\})], \{m, c\})$
- After Max binding
  $([(\langle s \rangle_4,\ \{\text{Max} \rightarrow m, \text{C} \rightarrow c\})],$
  $\{m = (\text{proc}\{\$\ X\ Y\ Z\}\ \langle s \rangle_3\ \text{end}\ , \varnothing)\ , c\}\ )$

Execution examples (3)

$\langle s \rangle_1$ $\langle s \rangle_2$

```
local Max C in
   proc {Max X Y Z}
⟨s⟩3  if X >= Y then Z=X else Z=Y end
   end
⟨s⟩4 {Max 3 5 C}
end
```

- After Max binding
  $([(\langle s \rangle_4, \{Max \rightarrow m, C \rightarrow c\})],$
  $\{m = (\text{proc}\{\$\ X\ Y\ Z\}\ \langle s \rangle_3\ \text{end}, \varnothing), c\})$
- After procedure call
  $([(\langle s \rangle_3, \{X \rightarrow t_1, Y \rightarrow t_2, Z \rightarrow c\})],$
  $\{m = (\text{proc}\{\$\ X\ Y\ Z\}\ \langle s \rangle_3\ \text{end}, \varnothing), t_1{=}3, t_2{=}5, c\})$

Execution examples (4)

$\langle s\rangle_1$ { $\langle s\rangle_2$ { $\langle s\rangle_4$

```
local Max C in
    proc {Max X Y Z}
    ⟨s⟩3  if X >= Y then Z=X else Z=Y end
    end
⟨s⟩4 {Max 3 5 C}
end
```

- After procedure call
  ([($\langle s\rangle_3$, {X → t_1, Y → t_2, Z → c})],
  {m = (proc{$ X Y Z} $\langle s\rangle_3$ end , ∅) , t_1=3, t_2=5, c})
- After T = (X>=Y)
  ([($\langle s\rangle_3$, {X → t_1, Y → t_2, Z → c, T → t})],
  {m = (proc{$ X Y Z} $\langle s\rangle_3$ end , ∅) , t_1=3, t_2=5, c, t=false})
- ([(Z=Y , {X → t_1, Y → t_2, Z → c, T → t})],
  {m = (proc{$ X Y Z} $\langle s\rangle_3$ end , ∅) , t_1=3, t_2=5, c, t=false})

Execution examples (5)

$\langle s \rangle_1$ $\langle s \rangle_2$

```
local Max C in
   proc {Max X Y Z}
    ⟨s⟩3  if X >= Y then Z=X else Z=Y end
   end
⟨s⟩4 {Max 3 5 C}
end
```

- ([(Z=Y , $\{X \rightarrow t_1, Y \rightarrow t_2, Z \rightarrow c, T \rightarrow t\}$)],
  $\{m = (\text{proc}\{\$\ X\ Y\ Z\}\ \langle s \rangle_3\ \text{end}, \varnothing),\ t_1{=}3,\ t_2{=}5,\ c,\ t{=}\text{false}\}$)
- ([ ],
  $\{m = (\text{proc}\{\$\ X\ Y\ Z\}\ \langle s \rangle_3\ \text{end}, \varnothing),\ t_1{=}3,\ t_2{=}5,\ c{=}5,\ t{=}\text{false}\}$)

Procedures with external references

$\langle s \rangle_1$ $\langle s \rangle_2$ $\langle s \rangle_3$

```
local LB Y C in
  Y = 5
  proc {LB X Z}
    if X >= Y then Z=X else Z=Y end
  end
  {LB 3 C}
end
```

Procedures with external references

$\langle s \rangle_1$ $\langle s \rangle_2$ $\langle s \rangle_3$

```
local LB Y C in
  Y = 5
  proc {LB X Z}
    if X >= Y then Z=X else Z=Y end
  end
  {LB 3  C}
end
```

- The procedure value of LB is
- (proc {\$ X Z} $\langle s \rangle_3$ end , {Y $\rightarrow$ y})
- The store is $\{y = 5, \ldots\}$

Procedures with external references

$\langle s \rangle_1$ / $\langle s \rangle_2$ / $\langle s \rangle_3$:

```
local LB Y C in
   Y = 5
   proc {LB X Z}
      ⟨s⟩3    if X >= Y then Z=X else Z=Y end
   end
   {LB 3 C}
end
```

- The procedure value of LB is
- (proc{\$ X Z} $\langle s \rangle_3$ end , {Y → y})
- The store is {y = 5, …}
- STACK: [({LB T C} , {Y → y , LB → lb, C → c, T → t})]
- STORE: {y = 5, lb = (proc{\$ X Z} $\langle s \rangle_3$ end , {Y → y}) , t = 3, c}

Procedures with external references

```
⟨s⟩1   local LB Y C in
         Y = 5
  ⟨s⟩2   proc {LB X Z}
    ⟨s⟩3   if X >= Y then Z=X else Z=Y end
         end
         {LB 3 C}
       end
```

- STACK: [({LB T C} , {Y → y , LB → lb, C → c, T → t})]
- STORE: {y = 5, lb = (proc{$ X Z} ⟨s⟩$_3$ end , {Y → y}) , t = 3, c}
- STACK: [(⟨s⟩$_3$, {Y → y , X → t , Z → c})]
- STORE: {y = 5, lb = (proc{$ X Z} ⟨s⟩$_3$ end , {Y → y}) , t = 3, c}

Procedures with external references

```
                local LB Y C in
                  Y = 5
⟨s⟩1   ⟨s⟩2        proc {LB X Z}
                     ⟨s⟩3   if X >= Y then Z=X else Z=Y end
                    end
                    {LB 3 C}
                end
```

- STACK: $[(\langle s\rangle_3, \{Y \rightarrow y, X \rightarrow t, Z \rightarrow c\})]$
- STORE: $\{y = 5, lb = (\text{proc}\{\$ \ X \ Z\} \ \langle s\rangle_3 \ \text{end}, \{Y \rightarrow y\}), t = 3, c\}$
- STACK: $[(Z{=}Y, \{Y \rightarrow y, X \rightarrow t, Z \rightarrow c\})]$
- STORE: $\{y = 5, lb = (\text{proc}\{\$ \ X \ Z\} \ \langle s\rangle_3 \ \text{end}, \{Y \rightarrow y\}), t = 3, c\}$
- STACK: [ ]
- STORE: $\{y = 5, lb = (\text{proc}\{\$ \ X \ Z\} \ \langle s\rangle_3 \ \text{end}, \{Y \rightarrow y\}), t = 3, c = 5\}$

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

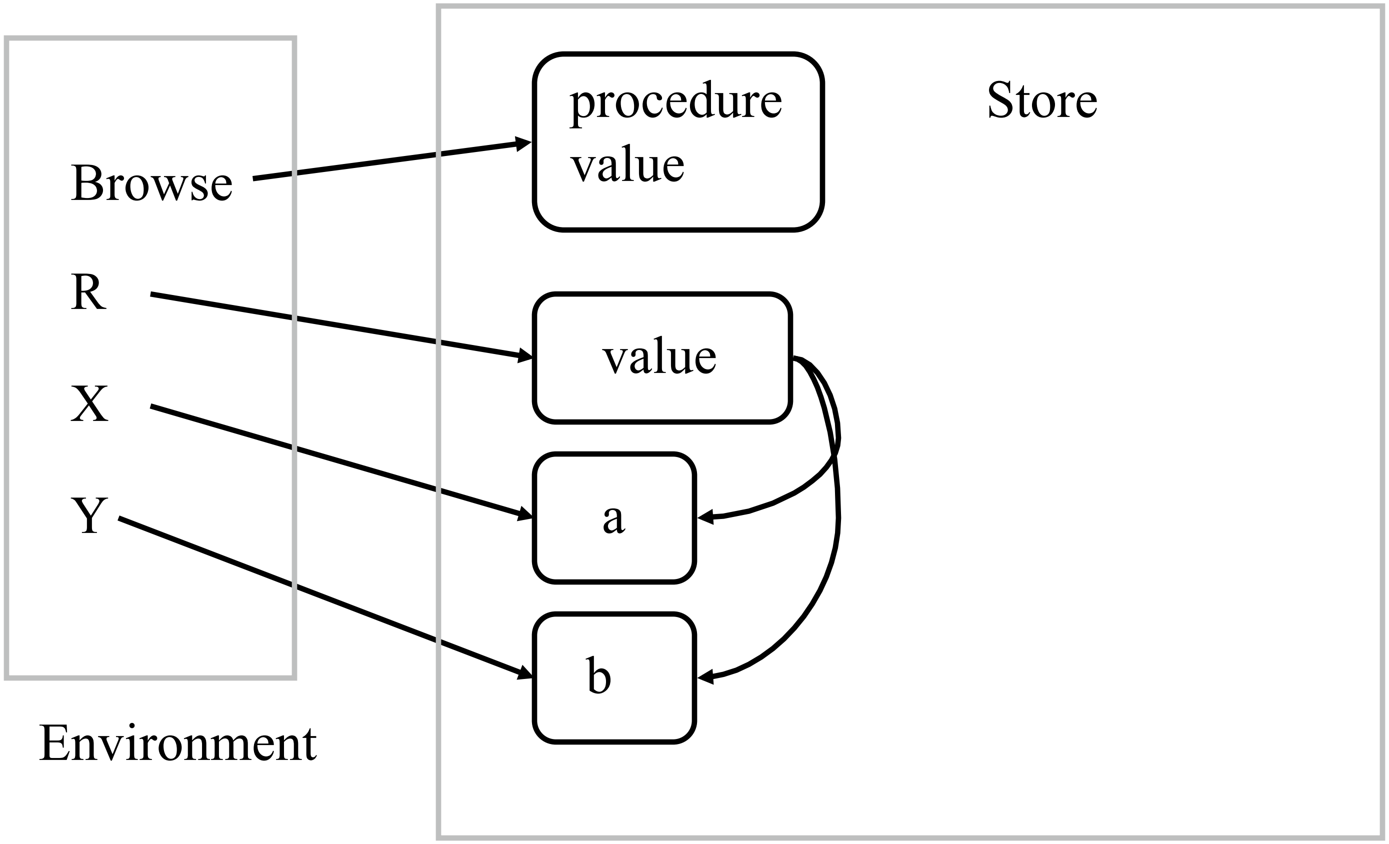
```
declare X Y
```

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

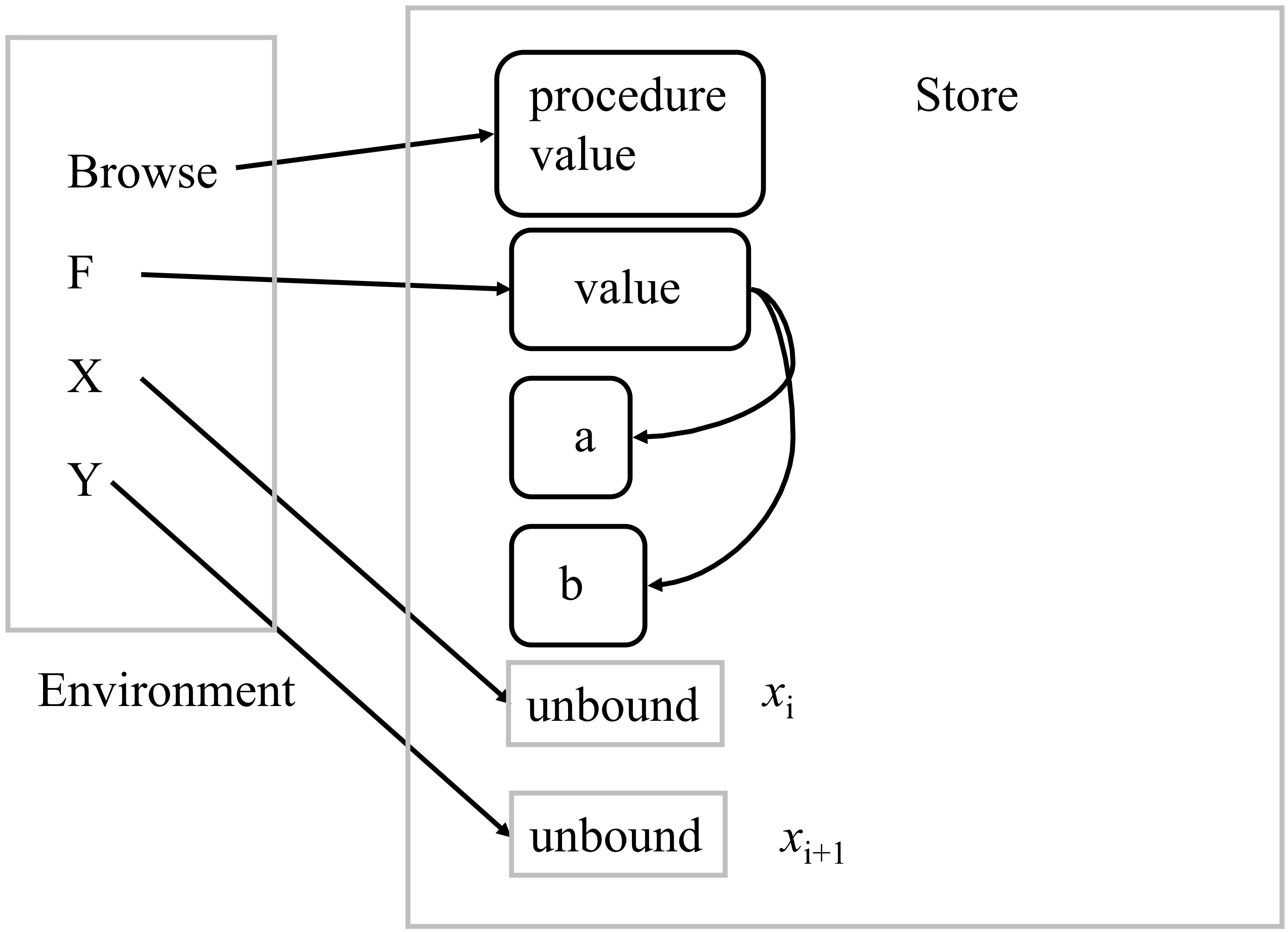
```
{Browse X}
```

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

The interactive interface (declare)



declare X Y


Browse
F
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

```
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)
- 11*11 X=11*11

expression

statement

Functions as linguistic abstraction

- R = {F X1 ... Xn}

↓

- {F X1 ... Xn R}

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

⟨statement⟩

→

```
proc {F 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

```
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 ({F X1 ... Xn})

  statement ({Q {F X1 ... Xn} ...})
  ```

Conditional expressions

```
R = if ⟨expr⟩1 then
        ⟨expr⟩2
    else ⟨expr⟩3 end
```

⟨expression⟩

→

```
if ⟨expr⟩1 then
   R = ⟨expr⟩2
else R = ⟨expr⟩3 end
```

⟨statement⟩

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

```
proc {Max 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
```

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

```
proc {Max 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 $\langle \text{expr} \rangle_1$ then
    $\langle \text{expr} \rangle_2$
else false end

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

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

Function calls

Observe

```
{F1 {F2 X} {F3 Y}}
```

⇒

```
local R1 R2 in
   R1 = {F2 X}
   R2 = {F3 Y}
   {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

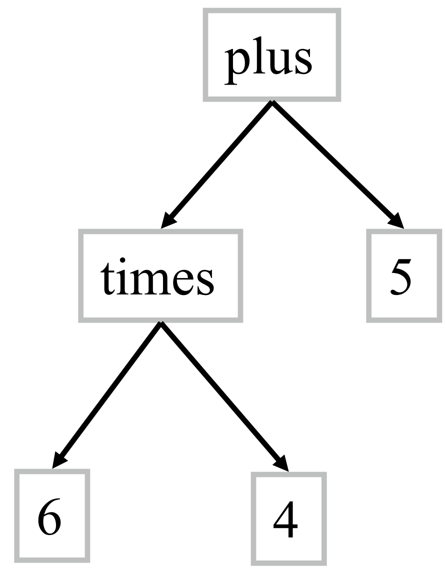
- The error confinement principle:
  - Define your program as a structured layers of components
  - Errors are visible only internally and a recovery procedure corrects the errors: either errors are not visible at the component boundary or are reported (nicely) to a higher level
- In one operation, exit 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 $\langle y \rangle$ then $\langle s \rangle_2$ end, E)
- Push the semantic statement (catch $\langle y \rangle$ 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 ⟨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 ⟨s⟩ end, E_c) be the catch statement that is found
- Push (⟨s⟩ , $E_c + \{<y> \rightarrow E(<x>)\}$) on ST
- Continue to next execution step

Catch semantics

- The semantic statement is
  (catch ⟨x⟩ then ⟨s⟩ 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
$\langle s \rangle_2$

Case statement revisited

- The semantic statement is
  (case ⟨x⟩ of ⟨l⟩ (⟨f_1⟩ : ⟨x_1⟩ … ⟨f_n⟩ : ⟨x_n⟩)
  then ⟨s_1⟩
  else ⟨s_2⟩ 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⟩ and its arity is [⟨f_1⟩ … ⟨f_n⟩]:
    push (try local ⟨x_1⟩ = ⟨x⟩. ⟨f_1⟩ … ⟨x_n⟩ = ⟨x⟩. ⟨f_n⟩ in ⟨s_1⟩* end
    catch user(E) then raise E end [] E then ⟨s_2⟩ end, E)
    on the stack, where ⟨s_1⟩* is the same as ⟨s_1⟩ except
    raise E end is changed to raise user(E) end
  - Otherwise, push (⟨s_2⟩, E) on the stack
- If the activation condition (E(⟨x⟩) is determined) is false:
  - Suspend

Exercises

49. VRH Exercise 2.9.3 (page 107).
50. VRH Exercise 2.9.7 (page 108) –translate example to kernel language and execute using operational semantics.
51. 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.
52. VRH 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.

Exercises

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