Declarative Computation Model

Rokas Gegevičius

September 28, 2022

Exercises

1. Free and bound identifiers. Consider the following statement:

```
proc {P X}
    if X>0 then {P X-1} end
end
```

   Is the second occurrence of the identifier **P** free or bound?

   Solution: let's translate it to the kernel syntax:

```
local P in
    local X in
        P=proc{$ X}
            local T in
                if T then {P X-1} end
            end
        end
    end
end
```

   P is bound to the procedure.

2. *Contextual environment.* Consider the following statement:

```
declare MulByN N in
N=3
proc {MulByN X ?Y}
    Y=N*X
end
```

   together with the call {MulByN A B} . Assume that the environment at the call contains $\{\ A \to 10, B \to x_1\}$. When the procedure body is executed, the mapping $\{N \to 3\}$ is added to the environment. Why is this a necessary step? In particular, would not $\{N \to 3\}$ already exist somewhere in the environment at the call? Would not this be enough to ensure that the identifier N already maps to 3?

Give an example where N does not exist in the environment at the call. Then give a second example where N does exist there, but is bound to a different value than 3.

Solution: according to the procedure definition, they carrier their own contextual environment

(**proc** {\$ $\langle y \rangle_1$, ..., $\langle y \rangle_n$} $\langle s \rangle$ **end**, CE)

with all the mappings of identifiers to the store to ensure vaidation of lexical scoping rules. This is because a procedure that works when it is defined will continue to work, independent of the environment where it is called.

Examples:

- N does exist in the environment at the procedure call

```
local MulByN in
    local N in
       N=3
       proc {MulByN X ?Y}
           Y=N*X
       end
    end
    {Browse {MulByN 10}} % prints 30
    {Browse N} % error does not exist
end
```

- N does not exist in the environment at the call

```
local MulByN N in
    N=3
    proc {MulByN X ?Y}
        Y=N*X
    end
    {Browse N} % prints 3
    {Browse {MulByN 10}} % prints 30
end
```

3. *Functions and procedures.* If a function body has an if statement with a missing else case, then an exception is raised if the if condition is false. Explain why this behavior is correct. This situation does not occur for procedures. Explain why not.

   Solution:

   procedure definition: **proc** {\$ $\langle X \rangle_1$, ..., $\langle X \rangle_n$} $\langle$*statement*$\rangle$ **end**

   function definition: **fun** {\$ $\langle X \rangle_1$, ..., $\langle X \rangle_n$} $\langle$*statement*$\rangle$ $\langle$*expression*$\rangle$ **end**

```
local F in
    fun {F X}
        if X==0 then 0 else 1 end
    end
end
```

   according to the function definition, it must end with an expression. If the else-case is missing and the if-condition is false, there is no expression at the end - raising an exception makes sense

```
local F in
    proc {F X ?R}
        if X==0 then R=0 else R=1 end
    end
end
```

   according to the definition of the procedure, it must end with a statement. If the else-case is missing and the if-condition is false, the R identifier will be unbound.

4. *The if and case statements.* This exercise explores the relationship between the if statement and the case statement.

   (a) Define the if statement in terms of the case statement. This shows that the conditional does not add any expressiveness over pattern matching. It could have been added as a linguistic abstraction.

   (b) Define the case statement in terms of the if statement, using the operations Label , Arity , and '.' (feature selection).

   This shows that the if statement is essentially a more primitive version of the case statement.

Solution:
(a) part

```
local Test
    fun {Test X}
        case X of 0 then 1
        else 0
        end
    end
in
    {Browse {Test 0}} % prints 1
    {Browse {Test 1}} % prints 0
end
```

(b) part

```
local
    R=test(x:value1 y:value2 z:value3)
in
    if {Label R}\=test then skip
    else if {Arity R}\=[x y z] then skip
        else if R.x == R.y then skip
            else {Browse 'Success: all cases were false'} end
        end
    end
end
```

5. *The* ***case*** *statement.* This exercise tests your understanding of the full case statement. Given the following procedure:

```
proc {Test X}
    case X
    of a|Z then {Browse 'case'(1)}
    [] f(a) then {Browse 'case'(2)}
    [] Y|Z andthen Y==Z then {Browse 'case'(3)}
    [] Y|Z then {Browse 'case'(4)}
    [] f(Y) then {Browse 'case'(5)}
    else {Browse 'case'(6)} end
end
```

Without executing any code, predict what will happen when you feed

```
{Test [b c a]}, {Test f(b(3))}, {Test f(a)}, {Test f(a(3))},
{Test f(d)}, {Test [a b c]}, {Test [c a b]}, {Test a|a},
and {Test '|'(a b c)}.
```

Solution:

```
Test [b c a] → case(4)
Test f(b(3)) → case(5)
Test f(a)    → case(2)
Test f(a(3)) → case(5)
Test f(d)    → case(5)
Test [a b c] → case(1)
Test [c a b] → case(4)
Test a|a,    → case(1)
Test'|'(a b c) → case(6)
```

6. *The **case** statement again.* Given the following procedure:

```
proc {Test X}
    case X of f(a Y c) then {Browse 'case'(1)}
    else {Browse 'case'(2)} end
end
```

Without executing any code, predict what will happen when you feed:

```
declare X Y {Test f(X b Y)}

declare X Y {Test f(a Y d)}

declare X Y {Test f(X Y d)}
```

Use the kernel translation and the semantics if necessary to make the predictions. After making the predictions, check your understanding by running the examples in Mozart. Now run the following example:

```
declare X Y
if f(X Y d)==f(a Y c) then {Browse 'case'(1)}
else {Browse 'case'(2)} end
```

Does this give the same result or a different result than the previous example? Explain the result.

Solution: conditionals, pattern machings and procedure aplications are suspendable statements

```
declare X Y {Test f(X b Y)}
declare X Y {Test f(X Y d)}
```

at call X is unbound execution suspended - waiting for X to be bound

```
declare X Y {Test f(a Y d)}
```

'case'(2)

```
if f(X Y d)==f(a Y c) then {Browse 'case'(1)}
```

different result

```
declare X Y Test f(X b Y) → 'case'(2)
declare X Y Test f(a Y d) → 'case'(2)
declare X Y Test f(X Y d) → 'case'(2)
```

X is unbound, but no matter what value it gets, these tuples are not equal: conditional can be evaluated

7. *Lexically scoped closures.* Given the following code:

```
declare Max3 Max5
proc {SpecialMax Value ?SMax}
    fun {SMax X}
        if X>Value then X else Value end
    end
end
{SpecialMax 3 Max3}
{SpecialMax 5 Max5}
```

   Without executing any code, predict what will happen when you feed:

```
{Browse [{Max3 4} {Max5 4}]}
```

   Solution: [4, 5] Hint:procedure aplication is a suspendable statment

8. *Control abstraction.* This exercise explores the relationship between linguistic abstractions and higher-order programming.

   (a) Define the function AndThen as follows:

```
fun {AndThen BP1 BP2}
    if {BP1} then {BP2} else false end
end
```

   Does the call

   {AndThen fun {\$} ⟨*expression*⟩$_1$ end fun {\$} ⟨*expression*⟩$_2$ end}

   give the same result as ⟨*expression*⟩$_1$ **andthen** ⟨*expression*⟩$_2$? Does it avoid the evaluation of ⟨*expression*⟩$_1$ in the same situations?

(b) Write a function OrElse that is to **orelse** as AndThen is to **andthen**. Explain its behavior.

Solution: The ability to pass functions as arguments is known as higher-order programming

(a) part

```
local
    fun{AndThen BP1 BP2}
        if {BP1} then {BP2} else false end
    end
in
    {Browse {AndThen fun {$} a==a end fun{$} 'success' end}}
    % prints success
end
```

is the same as

```
local
    fun {Test}
        a==a andthen 'success'
    end
in
    {Browse {Test}} % prints success
end
```

$\langle expression \rangle_2$ is not evaluated if $\langle expression \rangle_1$ is false

(b) part

```
local
    fun{OrElse BP1 BP2}
        if {BP1} then true else {BP2} end
    end
in
    {Browse {OrElse fun {$} a==b end fun{$} 'success' end}}
    % prints success
end
```

is the same as

```
local
    fun {Test}
        a==b orelse 'success'
    end
in
    {Browse {Test}} % prints success
end
```

$\langle expression \rangle_2$ is not evaluated if $\langle expression \rangle_1$ is true

9. *Tail recursion.* This exercise examines the importance of tail recursion, in the light of the semantics given in the chapter. Consider the following two functions:

```
fun {Sum1 N}
    if N==0 then 0 else N+{Sum1 N-1} end
end

fun {Sum2 N S}
    if N==0 then S else {Sum2 N-1 N+S} end
end
```

Now do the following:

(a) Expand the two definitions into kernel syntax. It should be clear that **Sum2** is tail recursive and **Sum1** is not.

(b) Execute the two calls {Sum1 10} and {Sum2 10 0} by hand, using the semantics of this chapter to follow what happens to the stack and the store. How large does the stack become in either case?

(c) What would happen in the Mozart system if you would call {Sum1 100000000} or {Sum2 100000000 0}? Which one is likely to work? Which one is not? Try both on Mozart to verify your reasoning.

Solution: (a)

```
local Sum1 in
    Sum1 = proc {$ N ?R}
        if N==0 then R=0
        else local N1 in
                 N1=N-1
                 local R1 in
                     {Sum1 N1 R1}
                         R=N+R1
             end
          end
       end
   end
end

local Sum2 in
    Sum2 = proc {$ N S ?R}
        if N==0 then R=S else
                         local N1 in
                         N1=N-1
                             local S1 in
                                 S1=S+N
                                 R={Sum2 N1 S1}
                             end
                         end
                     end
                 end
             end
```

(b)

{Sum1 10}

$([\{Sum1\ N1\ R1\}\{N \to n_0,\ R \to r_0,\ N1 \to n_1,\ R1 \to r_1\}]$
$[\{R = N + R1\}\{N \to n_1,\ R \to r_1,\ N1 \to n_2,\ R1 \to r_2\}]$
$[\{R = N + R1\}\{N \to n_2,\ R \to r_2,\ N1 \to n_3,\ R1 \to r_3\}]$
$[\{R = N + R1\}\{N \to n_3,\ R \to r_2,\ N1 \to n_4,\ R1 \to r_4\}]$
...
$[\{R = N + R1\}\{N \to n_9,\ R \to r_9,\ N1 \to n_{10},\ R1 \to r_{10}\}]$
$\{n_0 = 10, ..., n_{10} = 0, r_0, ..., r_{10} = 0\})$

stack grows with every recursive call!

{Sum2 10 0}

$([\{Sum2\ N1\ S1\}\{N \to n_0,\ S \to s_0,\ R \to r_0\ N1 \to n_1,\ S1 \to s_1\}]$
$([\{Sum2\ N1\ S1\}\{N \to n_1,\ S \to s_1,\ R \to r_1\ N1 \to n_2,\ S1 \to s_2\}]$
...
$([\{Sum2\ N1\ S1\}\{N \to n_9,\ S \to s_9,\ R \to r_9\ N1 \to n_{10},\ S1 \to s_{10}\}]$
$\{n_0 = 10, ..., n_{10} = 0, n_{11}, s_0 = 0, ..., s_{10}, s_{11}\})$

last call optimisation - constant stack size

(c)

{Sum1 100000000}:
stack grows with every recursive call, will exhaust allowed memory

{Sum2 100000000 0}:
tail recursion, constant stack, will return result

10. *Expansion into kernel syntax.* Consider the following function *SMerge* that merges two sorted lists:

```
fun {SMerge Xs Ys}
    case Xs#Ys
    of nil#Ys then Ys
    [] Xs#nil then Xs
    [] (X|Xr)#(Y|Yr) then
        if X=<Y then X|{SMerge Xr Ys}
        else Y|{SMerge Xs Yr} end
    end
end
```

Expand *SMerge* into the kernel syntax.

Solution:

```
local SMerge in
    SMerge = proc {$ Xs Ys ?R}
        case Xs of nil then R=Ys
        else
            case Ys of nil then R=Xs
            else
                case Xs of X|Xr then
                    case Ys of Y|Yr then
                        if X=<Y then R=X|{SMerge Xr Ys}
                        else R=Y|{Smerge Xs Yr} end
                    end
                end
            end
```

```
        end
    end
end
```

11. *Mutual recursion.* Last call optimization is important for much more than just recursive calls. Consider the following mutually recursive definition of the functions *IsOdd* and *IsEven* :

```
fun {IsEven X}
    if X==0 then true else {IsOdd X-1} end
end
fun {IsOdd X}
    if X==0 then false else {IsEven X-1} end
end
```

These functions are mutually recursive since each function calls the other. Mutual recursion can be generalized to any number of functions. A set of functions is mutually recursive if they can be put in a sequence such that each function calls the next and the last calls the first. Show that the calls {*IsOdd N*} and {*IsEven N*} execute with constant stack size for all non-negative *N*.

Sulution:
$([\{IsEven\ X1\}\{X \rightarrow x_n, X1 \rightarrow x_{n+1}\}]$
$([\{IsOdd\ X1\}\{X \rightarrow x_{n+1}, X1 \rightarrow x_{n+2}\}]$
...
...
$([\{IsEven\ X1\}\{X \rightarrow x_{n+k}, X1 \rightarrow x_{n+(k+1)}\}]$

In general, if each function in a mutually recursive set has just one function call in its body, and this function call is a last call, then all functions in the set will execute with their stack size bounded by a constant.

12. *Exceptions with a finally clause.* Section 2.7 shows how to define the **try / finally** statement by translating it into a **try / catch** statement. For this exercise, define another translation of

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

in which $\langle s \rangle_1$ and $\langle s \rangle_2$ only occur once. *Hint*: it needs a boolean variable.

Solution:

```
local Boolean E in
    try
        ⟨s⟩1
        catch X then E=X Boolean=true end
        ⟨s⟩2
        if Boolean==true then raise E end
    end
end
```

13. *Unification.* Section 2.8.2 explains that the bind operation is actually much more general than just binding variables: it makes two partial values equal (if they are compatible). This operation is called unification. The purpose of this exercise is to explore why unification is interesting. Consider the three unifications $X = [a\ Z]$, $Y = [W\ b]$ and $X = Y$. Show that the variables X, Y, Z, and W are bound to the same values, no matter in which order the three unifications are done.

    Solution:

```
declare R1 R2

local X Y W Z in
    X=[a Z]
    Y=[W b]
    X=Y
    R1=[X Y W Z]
end

local X Y W Z in
    Y=[W b]
    X=Y
    X=[a Z]
    R2=[X Y W Z]
end

{Browse R1==R2}
{Browse R1#R2}
```

X, Y, Z, and W are bound to the same values, no matter in which order the three unifications are done.