Declarative Computation Model Exercises

Annalise Tarhan

April 18, 2020

1 Free and Bound Identifiers

1.1 In the following statement, is the second occurrence of the identifier P free or bound?

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

The second occurrence of P is bound.

```
proc {P X} <s> end
```

is a syntactic shortcut for

```
P = proc {$ X} <s> end}
```

which binds P to a new procedure value. This procedure value is bound through the end of the statement, so the second occurrence of P must be bound as well. Additionally, once the omitted *declare* is added to the beginning of the statement, the Mozart system accepts the statement, which it wouldn't do if it contained an unbound variable identifier.

2 Contextual Environment

2.1 Consider the following statement and procedure call $\{MulByN\ A\ B\}$. Assume the environment at the call contains $A \rightarrow 10, B \rightarrow x_1$. When the procedure body is executed, $N \rightarrow 3$ is added to the environment. Why is this necessary? Wouldn't $N \rightarrow 3$ already exist somewhere in the environment at the call? Why is this not enough to ensure that N already maps to 3?

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

When the procedure body is executed, the environment contains only the procedure argument bindings, X and Y, and bindings that exist when the procedure is defined, N. This is a consequence of static scoping. A, B, and any N that exists in the calling environment are ignored.

2.2 Give an example where N does not exist in the environment at the call.

```
local A=10 B in
   {MulByN A B}
end
```

2.3 Give an example where N does exist there, but is bound to a different value than 3.

```
local A=10 B N=90
   {MulByN A B}
end
```

3 Functions and Procedures

3.1 If a function body has an if statement with a missing $else$ case, then an exception is raised if the if condition is false. Why? Why does this situation not occur for procedures?

Functions bodies must end with expressions. Procedures do not have this requirement. An equivalent procedure would accept another "return" variable identifier, which it would bind to the expression the function would end with. If the procedure doesn't bind the variable identifier to a value, it simply remains unbound.
Interestingly, the compiler will accept the statement with the missing $else$ case as long as the if expression evaluates to true, but it will not accept a statement like the following, regardless of the if expression's value.

```
fun {P X}
   if X>0 then 10 end
   15
end
```

By implication, functions must not only end with an expression, but if either the *then* or *else* part of an *if/then/else* statement is an expression, the other must be as well.

4 The *if* and *case* Statements

4.1 Define the *if* statement in terms of the *case* statement.

```
case X of true
   then <S1>
   else <S2>
end
```

4.2 Define the *case* statement in terms of the *if* statement.

```
if {Label X}==Lit andthen {Arity X}==[Feat1 ... FeatN]
      then local X1=X.Feat1 ... Xn=X.FeatN in
         <S1>
      else
         <S2>
end
```

5 The *case* Statement

```
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
```

5.1 What will {Test [b c a]} do?

*case*4 It is a list with a head and a tail. It doesn't match the first pattern because a lowercase a isn't a variable identifier, it is a value.

5.2 What will {Test [f(b(3))]} do?

*case*5 It is of the form $f(Y)$ where Y replaces $b(3)$. It doesn't match *case*2 because a is a value, not a variable.

5.3 What will {Test f(a)} do?

*case*2 It is an exact match. It doesn't match *case*5 because a *case* statement does not continue evaluating after it finds a match.

5.4 What will {Test f(a(3))} do?

*case*5 $a(3)$ does not evaluate to the character a.

5.5 What will {Test f(d)} do?

*case*5 For the same reason as $f(a(3))$

5.6 What will {Test [a b c]} do?

*case*1 It is a list where the first element is a.

5.7 What will {Test [c b a]} do?

*case*4 For the same reason as [b c a].

5.8 What will {Test a|a} do?

*case*1 It is written as a list pair, but is still a list whose first element is a.

5.9 What will {Test '|'(a b c)} do?

*case*6 It doesn't match any of the cases, so it falls to the *else* statement.

6 The *case* Statement Again

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

Note: I got a strange error when I tried to run the code, but I don't think that was the point of the exercise. It might have something to do with using a different version of Mozart. I answered the questions as best I could, but I also may have missed the point.

6.1 What will declare X Y {Test f(X b Y)} do?

*case*1 There are no direct matches, but also no contradictions. X is a variable, a is a character. b is a character, Y is a variable. Y is a variable, c is a character.

6.2 What will declare X Y {Test f(a Y d)}

*case*2 The third element of each is a character, but they aren't the same.

6.3 What will declare X Y {Test f(X Y d)} do?

*case*2 Again, the third element doesn't match.

6.4 Does the following code give the same result?

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

*case*2 This result is the same because d and c are still different characters.

7 Lexically Scoped Closures

```
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}
```

7.1 What will $\{Browse\ [\{Max3\ 4\}\ \{Max5\ 4\}]\}$ do?

When the function $SMax$ is defined each time, it "closes" the environment with the values that are present at the time. When $Max3$ is defined during the call $\{SpecialMax\ 3\ Max3\}$, it binds $Value$ to 3. When $\{SpecialMax\ 5\ Max5\}$ is called, it defines $Max5$ with $Value$ bound to 5. So, $\{Max3\ 4\}$ displays 4 and $\{Max5\ 4\}$ displays 5 since $4 > 3$ and $5 > 4$. Together, it looks like [4 5].

8 Control Abstraction

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

8.1 Does the call {*AndThen fun* {$} ⟨*exp*1⟩ *end fun* {$} ⟨*exp*2⟩ *end*} give the same result as ⟨*exp*1⟩ andthen ⟨*exp*2⟩? Does it avoid the evaluation of ⟨*exp*2⟩ in the same situations?

Yes. It gives the same result, but it evaluates ⟨*exp*2⟩ regardless of the value of ⟨*exp*1⟩. The purpose of *andthen* is to avoid the evaluation of ⟨*exp*2⟩ if ⟨*exp*1⟩ evaluates to false.

8.2 Write a function *OrElse* that is to *orelse* as *AndThen* is to *andthen*.

```
fun {OrElse BP1 BP2}
   if {BP1} then true else {BP2} end
end
```

9 Tail Recursion

```
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
```

9.1 Expand the two definitions into kernel syntax.

```
local Sum1 in
   Sum1=proc {$ N R}
      local T in
         local V in
            local W in
               T=(N==0)
               V=(N-1)
```

```
                        if T then R=0 else
                            W={Sum1 V $}
                            R=N+W
                        end
                    end
                end
            end
        end
end
```

```
local Sum2 in
    Sum2=proc {$ N S R}
        local T in
            local V in
                local W in
                    T=(N==0)
                    V=(N-1)
                    W=(N+S)
                    if T then R=S else R={Sum2 V W $}
                    end
                end
            end
        end
    end
end
```

9.2 Execute the two calls{Sum1 10} and {Sum2 10 0} and track what happens to the stack and the store. How large does the stack become?

$\{Sum1\ 10\}$

$([(\langle s\rangle,\ \varnothing)],\ \varnothing)$

$([(\langle s1\rangle,\ \{Sum1 \to s,\ A \to a_0,\ B \to b_0\})], \{s,\ a_0,\ b_0\})$

$([(\{Sum1\ A\ B\},\ \{Sum1 \to s,\ A \to a_0,\ B \to b_0\})],$
$\{s = (proc\{\$\ N\ R\}\ \langle s3\rangle\ end,\ \varnothing),\ a_0 = 10,\ b_0\}$

$([(\langle s3\rangle,\ \{N \to a_0,\ R \to b_0\})],$
$\{s = (proc\{\$\ N\ R\}\ \langle s3\rangle\ end,\ \varnothing),\ a_0 = 10,\ b_0\})$

$([(\langle s4\rangle,\ \{N \to a_0,\ R \to b_0,\ T \to t_0,\ V \to a_1,\ W \to b_1\})],$
$\{s = (proc\{\$\ N\ R\}\ \langle s3\rangle\ end,\ \varnothing),\ a_0 = 10,\ a_1 = 9\, b_0,\ b_1,\ t_0 = false\})$

First recursive call:
$([(\langle s4\rangle, \{N \to a_0, R \to b_0, T \to t_0, V \to a_1, W \to b_1\}), (\{Sum1\,A\,B\}, \{Sum1 \to s,\ A \to a_1,\ B \to b_1,\ \})],$
$\{s = (proc\{\$\ N\ R\}\ \langle s3\rangle\ end,\ \varnothing),\ a_0 = 10,\ a_1 = 9\,b_0,\ b_1,\ t_0 = false\}$

$([(\langle s4\rangle,\ \{N \to a_0,\ R \to b_0,\ T \to t_0,\ V \to a_1,\ W \to b_1\}),\ (\langle s3\rangle,\ \{N \to a_1,\ R \to b_1\})],$
$\{s = (proc\{\$\ N\ R\}\ \langle s3\rangle\ end,\ \varnothing),\ a_0 = 10,\ a_1 = 9\,b_0,\ b_1,\ t_0 = false\})$

$([(\langle s4\rangle,\ \{N \to a_0,\ R \to b_0,\ T \to t_0,\ V \to a_1,\ W \to b_1\}),\ (\langle s4\rangle,\ \{N \to a_1,\ R \to b_1,\ T \to t_1,\ V \to a_2,\ W \to b_2\})],$
$\{s = (proc\{\$NR\}\langle s3\rangle end, \varnothing), a_0 = 10, a_1 = 9a_2 = 8, b_0, b_1, b_2, t_0 = false, t_1 = false\})$

Second recursive call:
$([(\langle s4\rangle,\ \{N \to a_0,\ R \to b_0,\ T \to t_0,\ V \to a_1,\ W \to b_1\}),\ (\langle s4\rangle,\ \{N \to a_1,\ R \to b_1, T \to t_1, V \to a_2, W \to b_2\}),\ (\{Sum1\,AB\}, \{Sum1 \to s, A \to a_2, B \to b_2\})],$
$\{s = (proc\{\$NR\}\langle s3\rangle end, \varnothing), a_0 = 10, a_1 = 9a_2 = 8, b_0, b_1, b_2, t_0 = false, t_1 = false\})$

...

Because $N + \{Sum1\ V\ W\}$ cannot be completed until the recursive call is resolved, it stays on the stack until the end condition is reached. Each recursive call adds another of the same call to the stack, so the stack ends up having a size proportional to the number of recursive calls.

$\{Sum2\ 10\ 0\}$

$([(\langle s\rangle,\ \varnothing)],\ \varnothing)$

$([(\langle s1\rangle,\ \{Sum2 \to s,\ A \to a_0,\ B \to b_0,\ C \to c\})], \{s, a_0, b_0, c\})$

$([(\{Sum2\ A\ B\ C\}, \{Sum2 \to s,\ A \to a_0,\ B \to b_0,\ C \to c\})],$
$\{s = (proc\{\$\ N\ S\ R\}\langle s3\rangle end, \varnothing),\ a_0 = 10,\ b_0 = 0,\ c\})$

$([(\langle s3\rangle, \{N \to a_0,\ S \to b_0,\ R \to c\})],$
$\{s = (proc\{\$\ N\ S\ R\}\langle s3\rangle end, \varnothing),\ a_0 = 10,\ b_0 = 0,\ c\})$

$([(\langle s4\rangle,\ \{N \to a_0,\ S \to b_0,\ R \to c,\ T \to t_0,\ V \to a_1,\ W \to b_1\})],$
$\{s = (proc\{\$NSR\}\langle s3\rangle end, \varnothing), a_0 = 10, a_1 = 9, b_0 = 0, b_1 = 10, c, t_0 = false\})$

First recursive call:
$([(\{Sum2\,V\,W\ R\},\ \{V \to a_1,\ W \to b_1,\ R \to c\})],$
$\{s = (proc\{\$NSR\}\langle s3\rangle end, \varnothing), a_0 = 10, a_1 = 9, b_0 = 0, b_1 = 10, c, t_0 = false\})$

$([(\langle s3\rangle,\ \{N \to a_1,\ S \to b_1,\ R \to c\})],$
$\{s = (proc\{\$\, N\, S\, R\}\langle s3\rangle end, \varnothing),\ a_0 = 10, a_1 = 9, b_0 = 0, b_1 = 10, c, t_0 = false\})$

$([(\langle s4\rangle,\ \{N \to a_1,\ S \to b_1,\ R \to c,\ T \to t_1,\ V \to a_2,\ W \to b_2\})],$
$\{s = (proc\{\$\, N\, S\, R\}\langle s3\rangle end, \varnothing),\ a_0 = 10,\ a_1 = 9,\ a_2 = 8,\ b_0 = 0,\ b_1 = 10,\ b_2 = 19,\ c,\ t_0 = false,\ t_1 = false\})$

Second recursive call:
$([(\{Sum2\, V\, W\, R\},\ \{N \to a_1,\ S \to b_1,\ R \to c,\ T \to t_1,\ V \to a_2,\ W \to b_2\})],$
$\{s = (proc\{\$\, N\, S\, R\}\langle s3\rangle end, \varnothing),\ a_0 = 10,\ a_1 = 9,\ a_2 = 8,\ b_0 = 0,\ b_1 = 10,\ b_2 = 19,\ c,\ t_0 = false,\ t_1 = false\})$

$([(\langle s3\rangle, \{N \to a_2,\ S \to b_2,\ R \to c\})],$
$\{s = (proc\{\$\, N\, S\, R\}\langle s3\rangle end, \varnothing),\ a_0 = 10,\ a_1 = 9,\ a_2 = 8,\ b_0 = 0,\ b_1 = 10,\ b_2 = 19,\ c,\ t_0 = false,\ t_1 = false\})$

...

In this version, the stack stays at a constant size because the abstract machine uses last call optimization and this version uses a tail-recursive procedure.

9.3 What would happen in the Mozart system if you call {Sum1 100000000} or {Sum2 100000000 0}? Which one is likely to work? Which one is not?

Mozart takes a few seconds to calculate Sum2, eventually giving 5000000050000000. Sum1, on the other hand, crashes.

10 Expansion Into Kernel Syntax

```
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
```

10.1 Expand SMerge into kernel syntax.

```
proc {SMerge Xs Ys ?R}
   case '#'(Xs Ys) of '#'(nil Ys) then R=Ys
   else case '#'(Xs Ys) of '#'(Xs nil) then R=Xs
   else case '#'(Xs Ys) of '#'(X|Xr Y|Yr) then
      local T V in
         T=(X=<Y)
         R=X|V
         if T then {SMerge Xr Ys V}
            else {SMerge Xs Yr V}
         end
      end
   end
end
```

11 Mutual Recursion

```
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
```

11.1 Show that the calls {IsEven N} and {IsOdd N} execute with constant stack size for all non-negative N.

Not only is the stack size constant, it never exceeds one. If N=0, both IsEven and IsOdd return a boolean immediately with no further calls. For any other value, the call to IsEven or IsOdd is popped from the stack and replaced with a call to the other.

$([(\{IsEven\ 10\}, \varnothing)],\ \varnothing)$
$([(\{IsOdd\ X-1\}, \{X \to x_0\})],\ \{x_0 = 10\})$
$([(\{IsEven\ X-1\}, \{X \to x_1\})],\ \{x_0 = 10,\ x_1 = 9\})$
$([(\{IsOdd\ X-1\}, \{X \to x_2\})],\ \{x_0 = 10,\ x_1 = 9,\ x_2 = 8\})$
...

12 Exceptions with a Finally Clause

12.1 Define another translation of $try\,\langle s1\rangle\,finally\,\langle s2\rangle\,end$ in which $\langle s1\rangle$ and $\langle s2\rangle$ appear only once.

```
local CaughtException in
      try <s1> CaughtException=false
      catch X then CaughtException=true end
      <s2>
      if CaughtException then raise X end
      end
end
```

13 Unification

13.1 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.

If the three unifications are executed in the order given, X is bound to a list with two elements, a and an unbound variable identifier Z. Then, Y is bound to a different list with two elements, an unbound identifier W and b. Finally, X is bound to Y. During the unification process, the unbound identifier Z is bound to b and W is bound to a.

If the order of the first two bindings is reversed, the process is the same. If $X = Y$ is executed first, it is slightly different. First, X and Y are bound to each other and there is no information about their values. When one of the other two bindings is executed, the values that are bound to one are automatically bound to the other as well. Once the third binding occurs, all of the unbound identifiers have been bound to values.

In every case, W is bound to a, Z is bound to b, and X and Y are bound to lists containing two elements, a and b.