PLD Assignment 2

Matilde Broløs (jtw868)

March 11, 2020

A2.1

a)

Below the ned version of `applyFun` can be seen.

```
and applyFun (fnc, pars) localEnv =
      match fnc with
      | Symbol x when (List.contains x unops) ->
          match pars with
          | Cons (v, Nil) -> applyUnop x v
          | _ -> raise (Lerror ("Wrong number of arguments to " + x))
      | Symbol x when (List.contains x binops) ->
          match pars with
          | Cons (v1, Cons (v2, Nil)) ->
                applyBinop x (v1, v2) localEnv
          | _ -> raise (Lerror ("Wrong number of arguments to " + x))
      | Symbol x when (List.contains x varops) ->
          applyVarop x pars
        // applying a closure
      | Closure (Cons (Symbol "lambda", rules), closureEnv) ->
          tryRules rules pars closureEnv
      | Cons (Symbol "delta", rules) ->
          tryRules rules pars localEnv
      | _ -> raise (Lerror (showSexp fnc + " can not be applied as a function"))
```

b)

The dynamically scoped functions use deep binding, as it uses a stack of bindings between variable names and values. New variable pairs are pushed onto the stack, and are also popped when leaving the scope in which the variables existed.

c)

There will be no observable difference between shallow and deep binding, as we pass variables by value, not by reference, which means that no closure is built, and thus it is always the most recent value for the variable that we use.
We see that this is the case by looking at the `combine` function in the `RunLISP.fsx` file. Here we see that to combine two environments env1 and env2, we first check

whether env1 is empty, or if it contains a binding from x to v appended to a third environment. If the second is the case, we check if x is present in env2: if yes, and the value of x in env2 is equal to v, we append env3 to env2. If the value is not equal to v, then we can't combine the environments, and nothing happens. If x is not present in env2, we add (x, v) to env, and combine env3 and env2.

d)

A2.2

a)

An advantage is that there will be only one value for truth. If any non-zero values are considered true, then comparing two truth values can evaluate to false, as the values are technically not equal, even though they are both true.
Another advantage could be that it would avoid false positives, because only 1 would be considered as *true*. When there are fewer values that are considered as *true*, it is easier to predict and understand how to do calculations with truth values.
Disadvantages are that more checks would have to be implemented, to be certain that the input value is either 0 or 1. Furthermore, not all values would be accepted in an *if* or *while* statement, which means that we would need a way to deal with run-time errors, as opposed to the other version, where we can always evaluate values to either *true* or *false*.

b)

Advantages could be that an and-operation between the truth value `0xfff...ff` and an arbitrary value x will always return the value x, and likewise an or-operation between 0 and a value x will always return x.
Furthermore, the negation of the truth value `0xfff...ff` is equal to 0, and vice versa. This can be convenient when performing calculations.
A disadvantage is that multiple values evaluate to *true*.

c)

A suggestion to what values would be sensible for representing *true* and *false* could be 0 for *false* and all non-zero values for *true*. The implementation of logical conjunction and disjunction could look as follows.

```
(define and (lambda (0 0) (0)
                    (0 x) (0)
                    (x 0) (0)
                    (x x) (1)
            )
)

(define or (lambda (0 0) (0)
                   (0 x) (1)
                   (x 0) (1)
                   (x x) (1)
```

```
            )
)
```

Another suggestion could be `FALSE` for *false* and `TRUE` for *true*. The implementation of logical conjunction and disjunction could then look as follows.

```
(define and (lambda (FALSE FALSE) (FALSE)
                    (FALSE TRUE) (FALSE)
                    (TRUE FALSE) (FALSE)
                    (TRUE TRUE) (TRUE)
             )
)

(define and (lambda (FALSE FALSE) (FALSE)
                    (FALSE TRUE) (TRUE)
                    (TRUE FALSE) (TRUE)
                    (TRUE TRUE) (TRUE)
             )
)
```

These operations could be done variadic too, as it could be done as a recursive version of the binary solution. A proper return value when no arguments are given could then be *true*, as the recursion would then be easier to implement. Furthermore, it might make sense that `x and ()` returns *true* if x is true, thus () should evaluate to *true*.

A2.3

a)

A suggestion for a control structure to add to PLD LISP could be a while loop, as it makes structuring of code more intuitive, makes the program easier to understand, and is required for Turing completeness.
A suitable syntax could be `(while condition body)`, where the *condition* is the part that determines if the loop should run once more, and the *body* is the code that is run if the *condition* is true.
The second suggestion for a control structure could be exception handling, in the form of a try-with functionality. This way you would avoid errors when giving the wrong kind of input to functions, because it would not be necessary to make a case for each type of input.
A suitable syntax could be `(trywith expression exception)`, where the *expression* is the body that is performed, and the *exception* is the exception handler.

b)

A2.4

a)

First we check whether `upToLength` is linear. Lets write [1,17,11,5] as [3,17]++[11,5].
`upToLength`([3,17]++[11,5]) returns [1,2,3,4].

Now lets check whether that is the same as `upToLength`([3,17])++`upToLength`([11,5]).
`upToLength`([3,17])++`upToLength`([11,5]) = [1,2]++[1,2] = [1,2,1,2].
Thus the function is not linear.
Now we check whether the function is homomorphic. Lets first find iota $\iota = upToLength([]) = []$.

$$\begin{aligned} upToLength([3,17] + +[11,5]) &= upToLength([3,17]) \oplus upToLength([11,5]) \\ &= [1,2] \oplus [1,2] \end{aligned}$$

Thus $xs \oplus ys$ must be $xs + +map(lambday : length(xs) + y, ys)$.

b)

First we check whether `everyOther` is linear. Lets write ['e','v','e','r','y'] as ['e','v','e']++['r','y'].
`everyOther`(['e','v','e']++['r','y']) returns ['e','e','y'].
Now lets check whether that is the same as `everyOther`(['e','v','e'])++`everyOther`(['r','y']).
`everyOther`(['e','v','e'])++`everyOther`(['r','y']) = ['e','e']++['r'] = ['e','e','r'].
Thus the function is not linear.
Now we check whether the function is homomorphic. Lets first find iota $\iota = everyOther([]) = []$.

$$\begin{aligned} everyOther(['e,'v,'e'] + +['r,'y']) &= everyOther(['e,'v,'e']) \oplus everyOther(['r,'y']) \\ &= ['e,'e'] \oplus ['r'] \end{aligned}$$

There is no $\oplus$ that will give us the right answer, thus this function is not homomorphic.

c)

Again we check whether `returnEmpty` is linear.
`returnEmpty`([1,2]++[3,4]) = []
`returnEmpty`([1,2])++ `returnEmpty`([3,4]) = []
Thus the function is linear.
When a function is linear, it is also homomorphic. Here is why:

$$\begin{aligned} returnEmpty([]) &= [] \\ returnEmpty([1,2] + +[3,4]) &= returnEmpty([1,2]) \oplus returnEmpty([3,4]) \\ &= [] \oplus [] \end{aligned}$$

Thus if $\oplus$ is ++ the function is homomorphic.

d)

We check if the function is linear. `appendAll`([[1],[4,2]]++[[],[1]]) = [1,4,2,1].
`appendAll`([[1],[4,2]])++`appendAll`([[],[1]]) = [1,4,2,1]. Thus the function is linear.
Again, if a function is linear, it is also homomorphic. `appendAll`([])=[]=ι.

$$\begin{aligned} appendAll([[1],[4,2]] + +[[],[1]]) &= appendAll([[1],[4,2]]) \oplus appendAll([[],[1]]) \\ &= [1,4,2] \oplus [1] \end{aligned}$$

Thus the function is homomorphic with ι=[] and $\oplus$ is ++.

e)

Again we check whether the function is linear. `sorted`([1,3,3,7]) returns [1,7]. `sorted`([1,3])++`sorted`([3,7]) returns [1,3,3,7]. Thus the function is not linear.

$$\begin{aligned} sorted([]) &= [] \\ sorted([1,3]++[3,7]) &= sorted([1,3]) \oplus sorted([3,7]) \\ &= [1,3] \oplus [3,7] \\ sorted([1,3]++[2,7]) &= sorted([1,3]) \oplus sorted([2,7]) \\ &= [1,3] \oplus [2,7] \end{aligned}$$

Thus the function is homomorphic with

$$\oplus = \begin{cases} l1[0]++l2[1] & if\ l1[1] \leq l2[0] \\ l1[0] & otherwise \end{cases}$$

A2.5

a)

```
Unify(s=a(A,B),t=a(B,A))
Rule 6: c=a s1=A t1=B
              s2=B t2=A
  Unify(s=A, t=B)
    Rule 4: Bind A -> B
    Succeed
  Unify(s=B, t=A)
    Rule 3: u=B
    Unify(s=B, u=B)
      Rule 1: Succeed

Bindings:
A -> B

Resulting type: a(B,B)
```

b)

```
Unify(s=a(A,b(c)), t=a(B,A))
Rule 6: c=a s1=A    t1=B
              s2=b(c) t2=A
  Unify(s=A, t=B)
    Rule 4: Bind A -> B
    Succeed
  Unify(s=b(c), t=A)
    Rule 3: u=B
    Unify(s=b(c), u=B)
```

```
      Rule 5: Bind B -> b(c)
      Succeed

Bindings:
A -> B
B -> b(c)

Resulting type: a(b(c), b(c))
```

c)

```
Unify(s=a(A,b(B)), t=a(B,A))
Rule 6: c=a s1=A    t1=B
             s2=b(B) t2=A
  Unify(s=A, t=B)
    Rule 4: Bind A -> B
    Succeed
  Unify(s=b(B), t=A)
    Rule 3: u=B
    Unify(s=b(c), u=B)
      Rule 7: s contains u
      Fail
```

d)

```
Unify(s=a(b(c),b(B)), t=a(B,A))
Rule 6: c=a s1=b(c) t1=B
             s2=b(B) t2=A
  Unify(s=b(c), t=B)
    Rule 5: Bind B -> b(c)
    Succeed
  Unify(s=b(B), t=A)
    Rule 5: Bind A -> b(B)

Bindings:
B -> b(c)
A -> b(c)

Resulting type: a(b(c), b(b(c)))
```