PLD Assignment 2

Anders Lietzen Holst, wlc376

March 5, 2021

1 A2.1)

1.1 A2.1.a)

- *Find an S-expression containing a* `lambda` *which evaluates to* `0`*, but which, when changing that* `lambda` *to* `lambdaD`*, evaluates to* `1`*.*

Below is a snippet of `task1.le` of my code hand-in, showing the S-expression I found, as well as the version with one `lambda` changed to a `lambdaD`:

```
1  ; the value of this S-expression is 0.
2  (if (define x 0) ((lambda (f x) (f)) (lambda y x) 1))
3
4  ; this S-expression is identical to the expression above, except the innermost
5  ; 'lambda' has been changed to 'lambdaD'. the value of this S-expression is 1.
6  (if (define x 0) ((lambda (f x) (f)) (lambdaD y x) 1))
```

1.2 A2.1.b)

- *Is it possible to find an S-expression as above but where the* `lambda` *in question is a closed lambda expression? If possible, give an example; if not, argue why not.*

I would argue that it is impossible. Here is why:

If a particular lambda – let's call it `foo` – is a closed lambda expression, then the value of any variable occuring in the `foo`'s body **must** be associated with the corresponding lambda parameter variable - and thus a variable reference made within the body of `foo` can never point to a variable outside of the scope of `foo`.

Since there is no difference between static and dynamic scoping when there is only one scope of reference, changing `lambda` to `lambdaD` can never change the value of a closed S-expression.

1.3 A2.1.c)

- *Do dynamically scoped functions in PLD-LISP use shallow or deep binding?*

Deep binding. Here is why:

When a pattern p_i is matched to an expression e_i in `matchPattern`, each value in e_i is bound to a variable (or pattern) in p_i and these are pushed to the front of the environment (ie. variable stack). This is seen in the below snippet of `RunLISP.fsx` (relevant line highlighted):

```
1  and tryRules rules pars localEnv =
2    match rules with
3    | Cons (p, Cons (e, rules1)) ->
4        match matchPattern p pars with
5        | Some env -> eval e (env @ localEnv)
6        ...
7    ...
```

1.4 A2.1.d)

- *Does shallow vs. deep binding give any observable difference in PLD-LISP?*

No, it does not.

There are very rarely observable differences between using shallow and deep binding - according to PLDI, this can only occur when it is possible to take the address of a variable, and therefore also when using call-by-reference. However, this is irrelevant, since PLD-LISP uses pass-by-value as evidenced by the highlighted lines of code in the snippet below:

```
1  let rec eval s localEnv =
2    match s with
3    ...
4    | Cons (e1, args) -> // function application
5        applyFun (eval e1 localEnv, evalList args localEnv, localEnv)
6  ...
7  and evalList es localEnv =
8    match es with
9    ...
10   | Cons (e1, es1) -> Cons (eval e1 localEnv, evalList es1 localEnv)
```

2 A2.2

2.1 A2.2.a

- *What does the program print under call-by-reference, call-by-value, and call-by-value-result?*

In my answer, I will assume that that in the final ***`print`*** *statement, the operands to the two additions are evaluated* ***left to right****, and that the additions are also evaluated left-to-right.*

Under this assumption, the program prints "17" for **all three** parameter passing methods. Here is why:

If the two additions are evaluated left to right, then the temporary result of `u + v` will be stored in a temporary variable, which is not going to be modified by the call to `f` (when call-by-reference or call-by-value-result is used; under call-by-value all of this is irrelevant).

`f(u, v)` is, of course, always equal to `f(4, -2) = 15` (since, as explained, `u` and `v` are only modified inside `f`). Thus the final print statement is always equivalent to: `print(4 + (-2) + f(4, -2))`, which is equivalent to `print(17)`.

2.2 A.2.2.b)

- *i. What are the results of* ***`f(1)`*** *and* ***`g(1)`*** *as a function of* ***`n`*** *under call by need and call by name?*

All four combinations of functions and parameter passing methods return the same value of 2^{n+1}:

```
f(1) under call-by-need: 2^(n + 1)
f(1) under call-by-name: 2^(n + 1)
g(1) under call-by-need: 2^(n + 1)
g(1) under call-by-name: 2^(n + 1)
```

This is as expected, since there can be no observable (ie. semantic) difference between call by name/need in a functional language without side effects.

- *ii. In each of the four cases, how many additions are executed?*

In the program there are `(n + 1)` distinct additions - and this is the number of additions executed by `f` under both parameter passing methods.

At the same time, the function `g` builds a binary recursion tree of height `(n + 1)` - not simply `n` because we also have the last layer of `y = gn() + gn()`. However, we must subtract one to account for the root of the tree. Therefore, the number of additions

---

```
f(1) call by need: n + 1 additions
f(1) call by name: n + 1 additions
g(1) call by need: 2^(n + 1) - 1 additions
g(1) call by name: 2^(n + 1) - 1 additions
```

Why does call by need not decrease the number of additions to $O(n)$?

Answer: call-by-need only applies to function parameters, and it is wrong to think of call-by-need as a sort of memoization. However, if we were to rewrite `g` as such:

```
1  double x = x + x
2
3  g(x0) = let g1()  = x0 + x0         in
4          let g2()  = double (g1 ())  in
5          ...
6          let y     = double (gn ())
7          in  y
```

Then the `x` in `double` would only be evaluated once per call to `double`, even though it appears twice in the function body, and thus we would have the linear recursion tree and $O(n)$ complexity (whereas with call-by-name we would still have the binray recursion tree and exponential complexity).

---

3 A2.3

- *For each of the three functions* `concat`*,* `subtract`*, and* `altSum`*, determine whether this function is a map homomorphism, a homormorphism, or not a homomorphism.*

In classifying the functions, I will first check for map homomorphisms, then regular homomorphisms, and if both tests fail, then simply declare that the function is neither.

3.1 A2.3.a - `concat`

`concat` is a **map homomorphism**. To show this, I need to show the following three properties:

$$\text{concat } (x +\!\!+ y) = \text{concat}x +\!\!+ \text{concat}y$$
$$\text{concat } [x] = g(x)$$
$$\text{concat } [] = e$$

where $g(x)$ is some function and e is the neutral element for list concatenation. The latter is easy, since I know from PLDI that the neutral element for list concatenation is the empty list, ie. $e = []$.

Since concat $[x] = x$, the second property is satisfied by setting $g = \text{id}$ (the identity function).

Thirdly, I need to show the first property. This one is a little more tricky. I intend to show the property by intuition and argue that it is sufficient.

Let `x` and `y` be 2D lists of some element type (which may or may not in itself be a composite type); then `(x ++ y)` is a list containing `x`'s sublists followed by `y`'s sublists, and then `concat (x ++ y)` is a list containing the elements of `x`'s sublists followed by the elements of `y`'s sublists.

On the left hand side, `concat x` and `concat y` are two lists containing the elements of `x`'s sublists, and `y`'s sublists, respectively. Concatenating these with `(++)` yields a list containing the elements of `x`'s followed by the elements of `y`'s sublists.

We see that the two are equal. All three properties are satisfied, and thus `concat` is a map homomorphism.

3.2 A2.3.b - `subtract`

`subtract` is *not* a map homomorphism, because it has type `[int] -> int` and is thus not a function from lists to lists.

It is also disqualified from being a regular list homomorphism as there exists no binary operator $\odot$ which satisfies:

$$\text{subtract}\,(x +\!\!+\, y) = \text{subtract}\,(x)\ \odot\ \text{subtract}\,(y)$$

In conclusion, `subtract` is neither a map homomorphism nor a regular homomorphism.

3.3 A2.3.c - `altSum`

`altSum` is also *not* a map homomorphism, since it has type `[int] -> (int -> int)` (ie. takes a list of ints and returns a function which takes an int and returns an int).

It is, however, a regular list homomorphism. If we set g to be `altSum` itself, $\odot$ to be function composition, and e to be the identity function (since that is the neutral element for function composition), then the properties for a regular homomorphism are satisfied. Let's see how!

Consider the case where n is even. Then altSum$[x_1, \ldots, x_n]$ returns the function:

$$f(y) = x_1 - x_2 + x_3 - ... - x_n + y$$

Let then f_1 and f_2 be two functions given by:

$$f_1(y) = x_1 - x_2 + ... + x_{i-1} - x_i + y$$
$$f_2(y) = x_{i+1} - x_{i+2} + ... + x_{n-1} - x_n + y$$

where i is some even integer between 1 and n. Then the composition of f_1 and f_2 is precisely f, as seen below:

$$\begin{aligned}(f_1 \circ f_2)(y) &= x_1 - x_2 + \cdots + x_{i-1} - x_i + (x_{i+1} - x_{i+2} + \cdots + x_{n-1} - x_n + y) \\ &= f(y)\end{aligned}$$

Here I have only showed it for the case where both n and i are even, but the same also holds when both numbers are odd and when one is odd and one is even.

In conclusion, by setting $\odot = \circ$ (function composition), $g =$ `altSum`, and $e =$ id (identity function; neutral element for function composition), the properties for a regular list homomorphism are satisfied.

---

4 A2.4

4.1 A2.4a)

- *Which consequences does the introduction of the* `return` *statement have for reasoning about LISP programs?*

The extension has typing implications, and can eg. make it harder to reason about the return type of functions.

For example, consider the below function foo, which adds a number x to the head of a list:

```
(define foo
    (lambda (x myList)
        (+ x (head myList))
    )
)
```

Using Haskell-like type notation, `foo` has type `Num -> [Num] -> Num`, where `Num` is a numerical type (meaning it takes a number and a list of number and returns a number). However, if we introduce a `return` statement on `myList` like so:

```
(define foo
    (lambda (x myList)
        (+ x (head (return myList)))
    )
)
```

Then `foo` now has type `Num -> [Num] -> [Num]`, since `myList` is returned.

In any case, the return value of any given lambda will, of course, still be deterministic and thus predictable, and since PLD LISP is dynamically typed the `return` statement does not introduce any types of errors which are not entirely possible to make already - however, it might make it easier to screw up, especially when there are multiple `return` statements inside a single `lambda`, or when there are nested `lambda`s.

- *Before the extension, one could replace* `(+ x y)` *with* `(+ y x)` *inside any PLD-LISP expression that evaluates to a number without affecting semantics. Is this still the case after the introduction of the* `return` *statement?*

This question is ambiguous! In any case:

If `x` and `y` are variables, then this operation will *still* be commutative, since `x` and `y` will be fully evaluated before the addition operation, and any `return` statements encountered in evaluating `x` and `y` will not affect the addition operation.

However, if we take `x` and `y` to be arbitrary expressions, then the `return` statement can definitely change semantics - consider the below addition function:

```
1 (lambda (x y) (+ (return x) y))
```

Even though this function appears to be a wrapper for addition, which is a commutative operation, it is *not* commutative, since its value will always be that of its first parameter.

4.2 A.2.4.b)

- *How would you implement* `return` *in PLD-LISP?*

Note: I thought the task was to actually implement `return`*. The following section is probably a lot longer than it would have been if I had just described a sketch of the change; I hope the reader will bear with me.*

4.2.1 Disambiguating the `return` statement

There is some ambiguity in the specifications of the `return` statement; in particular, the assignment text does not state the semantics of `return` statements *outside* of lambda expressions. I thus propose the following two disambiguations:

Variant 1) simply make `return` statements outside of lambdas illegal. When a `return` statement is found outside a lambda, the interpreter should raise an "illegal return statement" exception.

Variant 2) ignore `return` statements outside of lambdas. When a `return` statement is encountered outside of a lambda, it is interpreted as the identity function (in other words, `(return x) = x` for any `x` when outside of a lambda).

I will attempt to implement both variants.

4.2.2 Implementing the two variants

I start by adding `"return"` to the set of unary operators, since that it what it is, and such that it becomes a reserved keyword:

```
let unops = ["number?"; "symbol?"; "return"]
```

Next, I take the hint given in the assignment text and use F# exceptions to implement the `return` statement. I add a new exception type:

```
exception ReturnStmException of Sexp
```

The new exception type contains an `Sexp`, which can be extracted and either returned or fed back into the `eval` function based on which variant is used.

Variant 1

The first variant is the simpler of the two. To implement it, I first extend `applyUnop` with a case for the new unary operator:

```
and applyUnop x v =
  match (x, v) with
  | ("return", s) -> raise (ReturnStmException s)
  ...
```

When a `return` statement is encountered, a "return" exception is thrown. This should of course be caught somewhere. If we are inside a `lambda`, then it should be caught, and the `s` inside should be evaluated. To do so, I simply modify the `lambda` case in `applyLambda`:

```
and applyLambda (fnc, pars, localEnv) =
  match fnc with
    | Closure (Cons (Symbol "lambda", rules), closureEnv) ->
        try tryRules rules pars closureEnv
        with
        | Lerror message -> ...
        | ReturnStmException s -> eval s localEnv
    ...
```

and similarly for the `lambdaD` case. Now, the value of a lambda containing a "`return x`" statement will simply be whatever `x` evaluates to.

If we are *not* inside a lambda, then we need to discard the entire expression currently being evaluated and report a "Return statement outside lambda" error. To do so, I catch the return exception at the top level of the REPL like so:

```
and repl infile () =
  ...
  match readFromStream infile "" with
  | Success (e, p) ->
     if p=0 then
       (try printf ...
        with
        | ReturnStmException foo -> printf "! Return statement outside lambda!\n"
        | Lerror message -> ... )
     ...
```

Variant 2

Variant 2 is a little more tricky. Since the `return` statement must now have different behavior based on the context in which it occurs, we need some way to keep track of whether or not we are currently inside a lambda.

I am not a very skilled F# programmer and do not know how to efficiently implement state in F#, so instead I simply extend all relevant functions with an "`inLambda`" flag which is true when evaluation is currently inside a lambda expression; else false.

The return case for `applyUnop` is then modified to only thrown the return exception when inside a lambda, and otherwise just treat the `return` operator as an identity function:

```
and applyUnop inLambda x v =
  match (x, v) with
  | ("return", s) -> if inLambda then raise (ReturnStmException s) else s
  ...
```

Below snippet then shows how `inLambda` is set to `true` when a lambda is evaluated, and how it is set to `false` at the beginning of evaluating an S-expression (relevant lines highlighted):

```
and applyFun inLambda (fnc, pars, localEnv) =
  match fnc with
    ...
    | Closure (Cons (Symbol "lambda", _), _) -> applyLambda true (fnc, pars, localEnv)
    | Cons (Symbol "lambdaD", _)             -> applyLambda true (fnc, pars, localEnv)

...

and repl infile () =
   ...
       (try
          printf "= %s\n" (showSexpIndent (eval false e []) 2 2)
        with
        | Lerror message -> ...)
        ...
```

This is of course a very tedious and hacky solution, since it involves extending every function declaration that is mutually recursive to `applyUnop`, as well as all calls to these functions, but it works nonetheless.

4.2.3 Testing the `return` statement implementations

I write two very simple test cases to test my implementation:

```
((lambda (x y) (+ x (return y))) 42 1337)

((lambda (x y) (+ x (return y))) (return 42) 1337)
```

The first test case should return 1337 for both variants, since the `return` statement is valid for both variants.

The second test case should fail for variant 1 (with a "*Return statement outside lambda!*" error message) since it contains a return outside of a lambda, and return 1337 for variant 2.

All tests are successful. To reproduce, run `make task4` while inside my code hand-in.

5 A2.5

5.1 A2.5.a

- *What should T be in the example given in the assignment text?*

`T` should be a function type from int to int, ie. `int -> int`, since the input array is an int array and the function operates on each element and writes back to the original array.

- *Suggest new type constructs that we should add to the type system for C in order to achieve higher order functions.*

This task is *highly* ambiguous because everything that is discussed in the assignment text is **already supported** by the C type system, and any (reasonable) C compiler (eg. gcc or clang) would, in fact, reject the program if `myarray` was anything but an integer array.

All this aside, there is undoubtedly **a lot** of room for improvement in the C type system for function types. In the following, I will discuss some features/constructs that would benefit the C type system.

Safety

For one, a lot of unsafety arises from the fact that C only supports function *pointers* and the fact that in C, any pointer can be typecast to a pointer of any different type. For this reason, it would be important to implement a distinct function type construct.

Facilitating optimization

Secondly, since one of the main benefits of programming in C is efficient memory utilization, it might be beneficial for the function type construct to include information about whether a particular function accesses global memory or not, and whether it both reads and writes or simply reads - in other words, whether or not the function can have side effects outside its own scope - as this would *significantly* increase the possibilities for automatic (memory) optimizations.

Ease of expression

As is, the syntactic side of the C function (pointer) type construct is baffling to say the least. As an example, consider the below Haskell program:

```haskell
h :: Int -> Int -> Int -> Int
h a b c = a + b * c

g :: Int -> Int -> Int -> (Int -> Int)
g a b c = h a (b + c)

f :: Int -> Int -> (Int -> Int -> (Int -> Int))
f a b = g (a + b)
```

Here, `f` has the type `Int -> Int -> (Int -> Int -> (Int -> Int))`. This is a rather obscure function type, but nevertheless it is not a particular complex type signature to discern for a human programmer.

In C function type notation, however, `f` has the type signature:

```
int (* (* (*)(int, int))(int, int))(int)
```

These kinds of ridiculous function type signatures make it very hard to implement and reason about higher-order functions in C. It would be a very good idea to design a more concise, expressive function type construct for the C type system, perhaps similar to that of Haskell.

However, unlike Haskell, C does not support partial application (or currying), and we should make this distinction clear in the syntax for the new function types. In general, the type signature for some function `g` of n parameters should in the source code be declared by:

```
foo :: (t_0, t_1, ..., t_n) -> t_return
```

In the case of the function `f` as described above, the type signature would in this system be:

```
f :: (int, int) -> ((int, int) -> ((int) -> int))
```

As such, the new syntax has some redundant parenthesization, but it is arguably a more concise syntax than the current syntax for function pointer types.

5.2 A2.5.b

- *Suggest how we should type check function declarations and function calls if we extend C with the previously discussed construct. The extend type system should handle cases such as the one presented in the assignment text.*

This question is a little ambiguous. It is not stated whether `T1, ..., T7` are all known at compile-time (as is required in C), or if the type system should support type inference. In the former case the answer is easy, because the C type system already supports everything necessary to type check the examples ;)

In the latter case, however, we would simply need to implement some sort of type inference system, since if the type signature of each function in a program can be inferred, then it must be well-typed.

In any case, we could type check functions as follows:

Given a function f with m parameters $(p_0, \ \ldots, \ p_{m-1})$ and function body f_{body}, we first look up its type signature in the type environment. If we find its type signature to be $f \ :: \ (t_0, \ ..., \ t_{n_1}) \rightarrow t_{\text{ret}}$ for some n (which may or may not be equal to m), then we type check the function f using the following steps (in order):

1. type check function arity (ie. that the function takes as many parameters as the type signature states): assert that $n = m$.
2. type check function parameters: assert that $p_i = t_i$ for $i \in \{0, \ldots, n\}$.
3. type check f_{body}: for each branch in the function body, assert that the value of this branch is t_{ret}.

---

6 A2.6

- *Give (at least) one example of slack in your favorite statically typed programming language that* does not *involve conditional statements or conditional expressions.*

My favorite statically typed programming language is, of course, C, and the kind of slack in the type system which I would like to discuss is that of dealing with different pointer types.

As we all know, C represents pointers as unsigned integers - when doing pointer arithmetic, the compiler uses the types of pointers to determine how much to add to a particular pointer (eg. if `xp` has type `int32_t*`, then the statement `xp += 1` will actually increment `xp` by 4 byte).

Below snippet shows a dummy program computing the size of an allocation by subtracting the distance between two pointers created by `malloc()`:

```
1  #include <stdlib.h>
2  #include <stdint.h>
3
4  #define ABS(x) (((long) (x)) < 0 ? -(x) : (x))
5  #define MAX(x, y) ((x) > (y) ? (x) : (y))
6
7  int main() {
8
9    // first, let's allocate some memory ...
10   size_t   request_size = 256 * sizeof(int);
11   int      *my_int_arr  = malloc(request_size);
12   unsigned *my_uint_arr = malloc(request_size);
13
14   // compute the distance between the two allocations.
15   int difference = my_int_arr - my_uint_arr;
16   int distance   = ABS(difference);
17
18   // to verify success of one of the allocations, assert that the
19   // distance between the two arrays is at least the size of the request.
20   int allocation_successful = distance >= request_size;
21
22   return allocation_successful;
23 }
```

The program is **incorrect** for a number of reasons, but that is not the matter of this analysis, because the program actually fails to compile due to a type checking error caused by line 15 (highlighted above). The type checker complains about invalid operands to binary minus since `my_int_arr` is an a signed int pointer, and `my_uint_arr` is an unsigned int pointer.

However, since both `int` and `unsigned` are both 32 bit large [1], the subtraction cannot produce errors, and type checker should be able to discern this since it knows the sizes of both types.

In any case, it is probably a good idea to be just a little conservative about pointer arithmetic - even for a language like C whose compilers does no take much responsibility for the programs they compile.

---

[1] Actually, `int` and `unsigned` are not guaranteed to be 32 bit, but they *are* guaranteed to be of the same size on a given machine.

7 A2.7

7.1 A2.7.a

- *Give an example of a function that cannot be written because of the requirement of array types, but which could still be written in C#.*

A good example is the "dynamic tables" examples that we all know and love from CLRS, in which the size of an array is doubled whenever it is filled up. Fixed upper bounds on statically declared Pascal arrays means the array can only be doubled a certain number of times - however, Pascal also supports dynamic arrays which can be used here.

7.2 A2.7.b

- *Suggest how to impose a bound on the length of arrays in C# that would be checked by the type system.*

The easiest solution is to create a new class for size-bounded arrays. In the snippet below I present a simple implementation for arrays of arbitrary number of dimensions:

```
1  public class MyBoundedArray<T> {
2    private readonly int[] bounds;
3    private Array array;
4
5    public MyBoundedArray(int[] bounds) {
6      this.bounds = bounds;
7      array = null;
8    }
9
10
11   public void initialize(int[] dims) {
12     if (dims.Rank != bounds.Rank) {
13       throw new ArgumentException("invalid dimensions");
14     }
15     for (int i = 0; i < bounds.Rank; i++) {
16       if (dims[i] > bounds[i]) {
17         throw new ArgumentException("Dimension " + i + " exceeds bound");
18       }
19     }
20     array = Array.CreateInstance(typeof(T), dims);
21   }
22
23   public void set(int[] idxs, T val) { array.SetValue(val, idxs);       }
24   public T    get(int[] idxs)        { return (T) array.GetValue(idxs); }
25
26   public int[] getBounds()  { return bounds;       }
27   public int getRank()      { return array.Rank;   }
28
29 }
```

Given an array of upper bounds for each dimension, a new array is *declared* (but not yet initialized) using the class constructor.

The important function here is then `initialize(dims)` (highlighted above), which (re-)initializes a new array in place of the old with the given dimensions *if* they are within the declared bounds; if they are not, or if an invalid number of dimensions are given, an exception is thrown.

7.3 A2.7.c

- *Are there good reasons why Pascal array types are the way they are?*

Yes! In Pascal, statically sized arrays allows the programmer to use a different indexing base.

To illustrate how this works and why it is useful, consider the problem of encoding a discrete two-dimensional real-valued function over a plane of discrete points centered around the origin (ie. the point $(0, 0)$). For the purposes of this example, consider the discrete plane of points (i, j) which satisfy $-50 <= i, j <= 50$ [2].

In most programming languages, we would use an array of dimensions `[101][101]`. The origin would have index `[50][50]` in this 2D array, while an arbitrary point (x, y) would have index `[x + 50][y + 50]`. Working with this 2D plane quickly becomes tedious!

However, using static Pascal arrays, we can declare the plane as such:

```
1  myPlane: array [-50 .. 50, -50 .. 50] of Real
```

This specifies `myPlane` to be a 101×101 array (since the intervals are inclusive) of reals, and that its indices range from -50 to $+50$ in either direction, rather than from 0 to 100 as discussed above. Now, any given point (x, y) in the plane (for which $-50 \leq x, y \leq 50$) has location `myPlane[x][y]` - even for negative coordinates.

[2] More formally the set $\{(i, j) \ : \ -50 \leq i \leq 50 \wedge -50 \leq j \leq 50\}$.