PLD Assignment 2

Anders Lietzen Holst, wlc376

March 16, 2021

1 A2.2

1.1 A.2.2.b.ii) - resubmission

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

In my feedback I was told that under call-by-name, the let bindings in `f` are evaluated like function calls. Here is my resubmission with the answers changed accordingly. New stuff is in written in red.

In the program there are (n + 1) distinct additions - and this is the number of additions executed under call-by-need, since here, the result of let bindings are memoized and not reevaluated, whereas with call-by-name, a binary recursion tree is built in evaluating the let bindings. The below table is changed accordingly.

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: 2^(n + 1) - 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 $O(n)$ for `g`?

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:

```
double x = x + x

g(x0) = let g1()  = x0 + x0         in
        let g2()  = double (g1 ())  in
        ...
        let y     = double (gn ())
        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).

---

2 A2.5

Here comes my resub of 2.5.b. New stuff is written in red.

2.1 A2.5.b - resubmission

- *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, ..., 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}.

- In the feedback received I was asked to determine the types of T1, ..., T7, and to explain what must hold for these in order for the program to be well-typed.

First, I write out the function types in Haskell-like type notation:

```
f :: T2 -> T3 -> T1
h :: T5 -> T4
j :: T7 -> T6
```

First off, from the definition of `f`, I can infer that `T2` is a function which takes a parameter of type `T3` and returns a `T1`, so I add the binding `T2 = T3 -> T1`.

Secondly, from the definition of `j`, I know that `f(h, v) :: T6`, which gives me the type binding `T6 = T1` since I know `f` to have return type `T1`.

Since `h` and `v` - which have types `T5 -> T4` and `T7`, respectively - are passed as parameters to `f`, which has type `T2 -> T3 -> T1`, I can also add the type bindings `T2 = T5 -> T4` and `T3 = T7`.

Lastly, since both `T2 = T5 -> T4` and `T2 = T3 -> T1`, I can infer that `T5 = T3` and `T4 = T1`.

In conclusion, I have inferred the following type bindings:

```
T2 = T3 -> T1
T2 = T5 -> T4
T6 = T1
T7 = T3
T5 = T3
T1 = T4
```

which can be reduced to:

```
T5 = T7 = T3
T4 = T6 = T1
T2 = T3 -> T1
```

As such, for the program to be well-typed, these are the restrictions which must hold. The function types given earlier can be redeclared as:

```
f :: T2 -> T3 -> T1
h :: T3 -> T1
j :: T3 -> T1
```

In a language with currying, the function types could of course be given as:

```
f :: T2 -> T2
h :: T2
j :: T2
```