Fixpoint Theory Notes – based on Zohar Manna's papers

These are meant as steps you can check off as you read about the Fixpoint Theory of programs. Kindly have two papers in front of you: `manna-fp-theory-1.pdf` and `manna-fp-theory-2.pdf`. These are under `papers/` on Canvas.

1. `[ ]` Some notations and simplification rules:
   - I inherit the notation below from Steven Johnson's excellent PhD dissertation from 1985 where he used recursion equations to compile into hardware circuits.

     $$(P \to Q, R)$$

     This means, if P then Q else R, where P, Q and R can themselves be conditionals.
   - Here are some rules for simplifying conditional expressions. Most of them are of the form of "if-lifting:"
     - "if-lifting" – pulling the predicates to the top level – is a handy simplification rule. In general, you can do this if you are careful about thinking about termination (you don't want the program after if-lifting to be less defined, i.e. "loop more often.")
     - This is one form of "if-lifting". Here "+" is for illustration; it can be any operator.

       $$(P \to Q, R) + X = (P \to Q + X, R + X)$$

     - This helps process nested conditionals:

       $$(P \to (P_1 \to Q_1, R_1), R)$$

       can simplify to

       $$(P \wedge P_1 \to Q_1,\ P \wedge \neg P_1 \to R_1,\ \neg P \to R)$$

   - There are many more such rules
2. `[ ]` The first thing I'll get "out of your way" is the recursive function `F` and the solutions for `F` in the form of three distinct functions `f1`, `f2`, and `f3`. All these are fixpoints of the `Tau` functional underlying `F`. Of these, `f3` is **the** least fixpoint. These functions defined in manna-fp-theory-1.pdf are reproduced below. Here on, I'm using `bottom` for the bottom (undefined) value and `BOTTOM` for the undefined (bottom) function. Recall that `BOTTOM(x) = bottom` for any `x`.

```
F(x,y) = (x=y) -> y+1, F(x, F(x-1, y+1))
f1(x,y) = (x=y) -> (y+1), (x+1)
f2(x,y) = (x>=y) -> (x+1), (y-1)
f3(x,y) = (x>=y) /\ even(x-y) -> (x+1), bottom
```

3. [ ] You can unfold `F` as a series of approximations. manna-fp-theory-1.pdf shows this in Section 1.2 for the simpler "Factorial" function. I'll show the unfolding for the aforesaid `F` function.

   ```
   F_0 = Tau^0[BOTTOM] = BOTTOM
   F_1 = Tau^1[BOTTOM] = Tau[Tau^0[BOTTOM]] = (x=y) -> y+1, bottom
   ```

4. [ ] Now,

   ```
   F_2 = Tau^2[BOTTOM] = Tau[Tau^1[BOTTOM]] =
   ```

   is obtained by substituting `Lambda x,y :  (x=y) -> y+1, bottom` in place of "F". We get

   ```
   (x=y) -> y+1,
   (x = [(x-1 = y+1) -> y+1+1, bottom]) ->          ; Detail (*) below
        [(x-1 = y+1) -> y+1+1, bottom] + 1 , bottom ; Detail (**) below
   ```

5. [ ] Now, focus on the line `(*)` above:

   ```
   (x = [(x-1 = y+1) -> y+1+1, bottom]) ->
   ```

   includes only one meaningful case in it, namely `x=y+1+1` or that `x=y+2`.

   In fact, we can simplify the above line to the nested conditional

   ```
   [(x-1 = y+1) -> [(x = y+1+1) -> ...  , ElseWhoCares] ..]
   ```

6. [ ] This is because all other cases involve `bottom`, and so `ElseWhoCares` are those bottom-cases.

7. [ ] Now, coming to (**), notice that the "y" field for the outer call is `[(x-1 = y+1) -> y+1+1, bottom]` and it is "that y" that has to be incremented by 1.

8. [ ] The expression `[(x-1 = y+1) -> (y+1+1), bottom] + 1`
   can be simplified to
   `[(x = y+2) -> (y+1+1)+1, (bottom+1)]`
9. [ ] Thus, together with (*) and (**), the whole can be simplified to
   `[(x = y+2) -> (y+3), bottom]`
10. [ ] So

    ```
    F_2 = Tau^2[BOTTOM] = Tau[Tau^1[BOTTOM]]
    ```

    simplifies to

    ```
    (x=y) -> y+1,
    (x = y+2) -> y+3, bottom
    ```

11. [ ] Now,

    `F_3 = Tau^3[BOTTOM] = Tau[Tau^2[BOTTOM]]`

    and by now, you have to "see" how this unraveling is going, and write the answer directly as below. The trick to "seeing" this answer is to focus on the inner-most recursion of `F` which is decreasing `x` and increasing `y` each time. This means that the final conditional that decides things is of the form x = y+4. Then once you land at that conditional, it is easy to see that the answer returned is y+5, otherwise it is "bottom everywhere". Thus we get

12. [ ]

    ```
    (x=y) -> y+1,
    (x = y+2) -> y+2, bottom
    (x = y+4) -> y+5, bottom
    ```

13. [ ] Now for context-free grammars being interpreted as a system of recursive equations. Consider the CFG

    `S -> aSbS | bSaS | epsilon`

If `L_S` denotes "the language of S," one can also view the aforesaid CFG production scheme being a language equation of the form shown below, following how we are supposed to read CFG rules right-hand sides (here `U` means union):

```
L_S  = {a} L_S {b} L_S   U   {b} L_S {a} L_S   U   {epsilon}
```

14. `[ ]` The above language equation has a unique solution, namely the least fixpoint, which is the set of all strings over $\{a, b\}$.

    In particular, $\{a, b\}^*$ cannot be a solution because the LHS when substituted with $\{a, b\}^*$ can contain, say, "bb", while the RHS can never contain a `b` without an `a`.

15. `[ ]` Introducing any redundancy such as

    ```
    S -> aSbS | bSaS | epsilon | S
    ```

    or

    ```
    S -> aSbS | bSaS | epsilon | SS
    ```

    allows us to obtain $\{a, b\}^*$ as another solution. This "extraneous" solution cannot be computed by following the CFG derivation rules—much like functions `f1` and `f2` mentioned earlier are "out of thin air" solutions—not modeling how programs laboriously construct the answer by following the strict recipe of recursive calls.

16. `[ ]` Why an extraneous solution? This is because we can substitute $\{a, b\}^*$ for the `S` on both sides of the equation. Then the Union operator blends the LHS and RHS into equality, by applying the union (`U`) operator.

17. `[ ]` Is mutual recursion any different, in terms of how we approach things? The answer is "no". We are solving for a *pair* (in general a tuple) of functions in those cases. Consider this mutual recursion:

    ```
    f1(x) = even(x) -> x+1, f2(x)
    f2(x) = odd(x) -> f1(x+2), x+1
    ```

It is clear that both `f1` and `f2` compute the same function: successor of evens if given an even input, otherwise (for odd inputs) being undefined. Here, we can Lambda-abstract and write it as a recursion:

```
(f1,f2) = [ Lambda G: (lambda x: even(x) -> x+1, snd(G)(x),
                       lambda x: odd(x) -> fst(G)(g), x+1 ) ] (f1,f2)
```

In other words, we are solving for a *pair* of functions `(f1,f2)` simultaneously. This is evident from the fact that the Tau functional we write has a `Lambda G` outside, where `G` has type "a pair of functions." This is why, inside the function body, we use `snd(G)` (second of `G`) and `fst(G)` (first of `G`).

18. `[ ]` Now we can do a fixpoint itertion as follows, to discover the "pair solution" (call it `f12` and let the approximants be `f12_0`, `f12_1`, etc.):

```
Tau =
[ Lambda G:  (lambda x: even(x) -> x+1, snd(G)(x),
                        lambda x: odd(x) -> fst(G)(g), x+1 )  ]

f12_0 = [ BOTTOM, BOTTOM ]  ; the pair of bottom functions

f12_1 = Tau(f12_0)

      = [ lambda x: even(x) -> x+1, BOTTOM(x),
          lambda x: odd(x) -> BOTTOM(g), x+1 )  ]

      = [ lambda x: even(x) -> x+1, bottom,
          lambda x: odd(x) -> bottom, x+1 )  ]

f12_2 = Tau(f12_1)

      = [ lambda x: even(x) -> x+1, snd(f12_1)(x),
          lambda x: odd(x) -> fst(f12_1)(g), x+1 )  ]
...
```

19. Now consider CFG productions that are mutually recursive. The same argument – modeling the languages as a pair – works. We iterate from `(BOTTOM, BOTTOM)` which, in case of languages, is $(\emptyset, \emptyset)$.

```
S -> ( W S | epsilon  ; I'll abbreviate epsilon as e, below.

W -> ( W W | )
```

20. `[ ]` The above can be solved as a pair of languages:

```
(L_S, L_W) = [  {(} L_W L_S  U  {e} ,
                {() L_W L_W  U  {)} ]
```

21. `[ ]` We iterate from `(, )` as "(BOTTOM, BOTTOM)" and this stabilizes to the pair language solution. Then from this solution, we can pick `L_S` as the first component of the solution fixpoint, and `L_W` as the second component of the solution fixpoint.