**Question 1**

i See notes. M, N :== x | \lambda x . M | MN

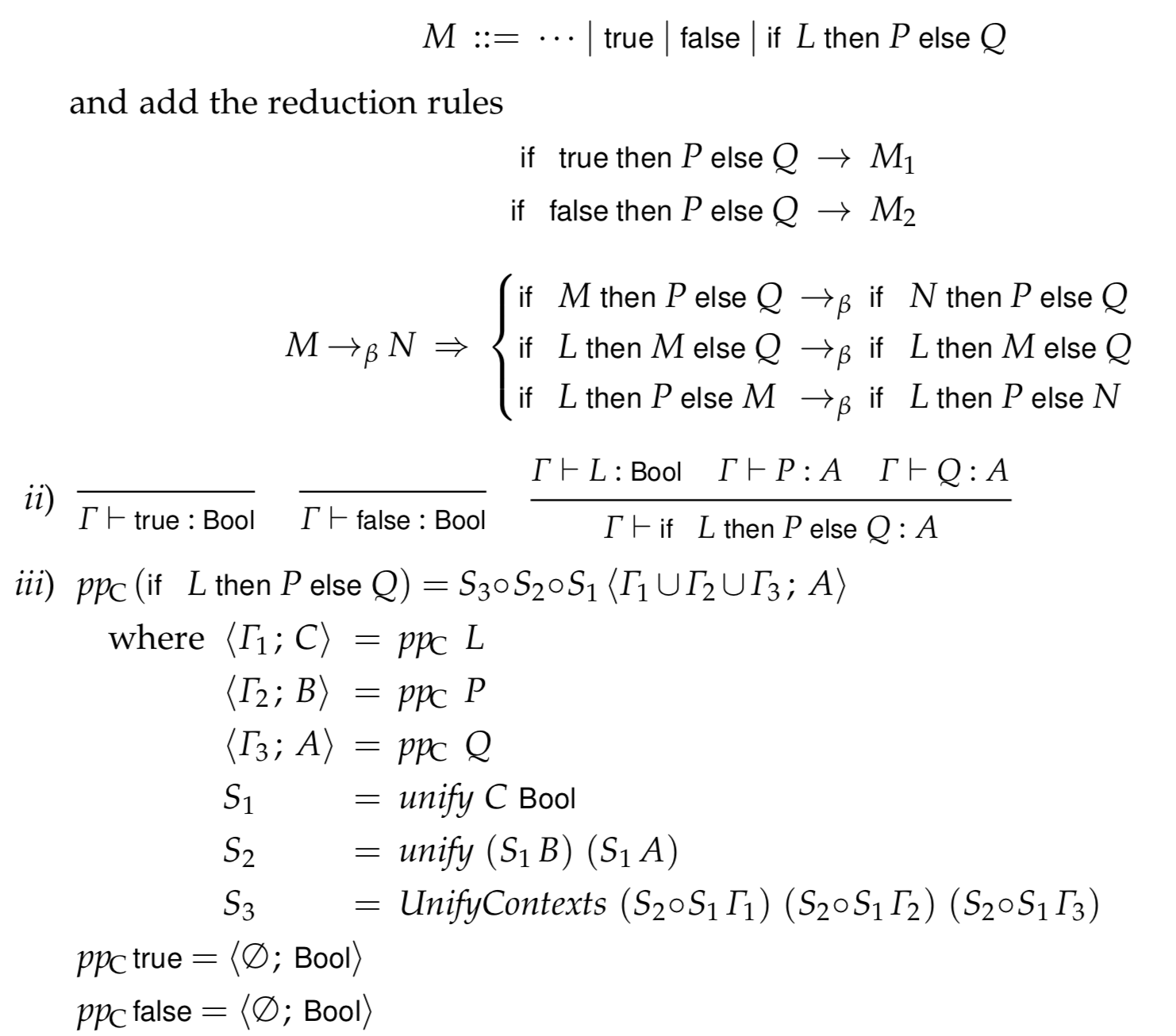
ii A, B :== \phi | A -> B

iii three cases correspond to the definition

b show that YM -> M(YM)

c

Exercise 2.20



d

Using Curry’s type assignment system you can only type terms that are strongly normalizable. However, when translating from haskell some terms would have reduction sequences that don’t lead to a normal form due to recursion. We can overcome this by using the Y-combinator to model recursion and adding the typing rule Gamma |- Y : (A->A)->A

e

No. See Exercise 2.24 ii

2

a

see notes

b

see notes

c

i) Yes. Let x be of type A ->A and y be of type A

ii) No. Can not unify the type of x with the type of (\lambda y.x)

iii) Yes. Let x be of type A->B and y be of type A.

d

fix Mult. \lambda x y . cond (=0 x) 0 (+ (Mult (-1 x ) y) y)

e

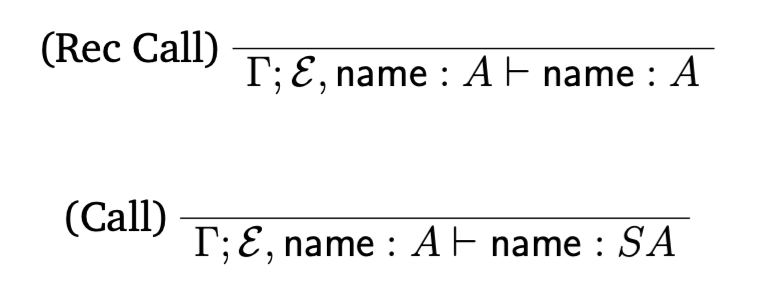
See notes ex 5.18



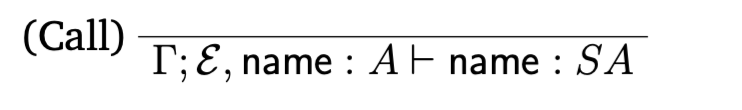
---

3d)

Milner has two type assignment rules:

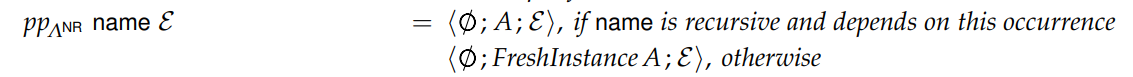


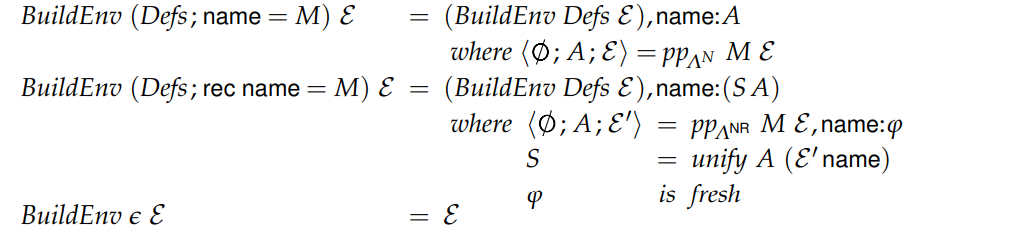
Mycroft only has one, even for recursive calls:



This means recursive calls can be polymorphic.

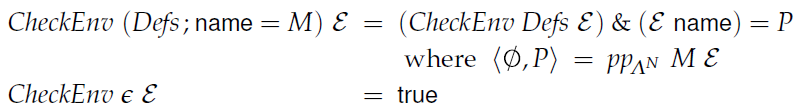
Milner’s pp/\NR algorithm when dealing with recursive calls:





Mycroft’s:





Mycroft’s pp/\NR algorithm checks the environment rather than constructs it (how is this related to type assignment rules???)

4) d) The answer to this is in the notes (Figure 11) but if you're stuck, hints:

- Use a case statement

- Think about what a list is represented by in equi-recursive approach (from notes: inj.l () for an empty list or inj.r <E1, E2> for E1 "cons" E2 ----- or even more generically, a list is represented by unit + (B X [B])).

4) a. def in notes

4) b. def in notes

4) c. derivation

4) d. Y (λf xs. case(xs, λxs’. 0, λxs’. (+) 1 (f (right xs’)))

Then the answer follows by the derivation, the first step in the tree requires the proof of

Ø |- Y : (([φ] -> Int) -> ([φ] -> Int)) -> ([φ] -> Int)