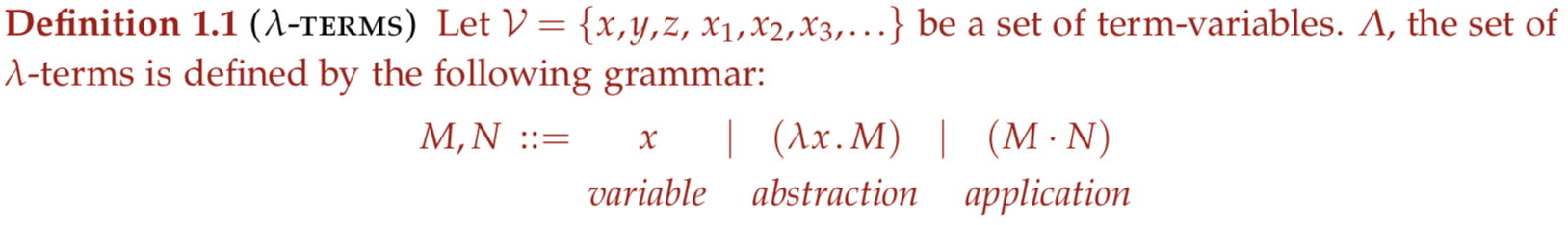
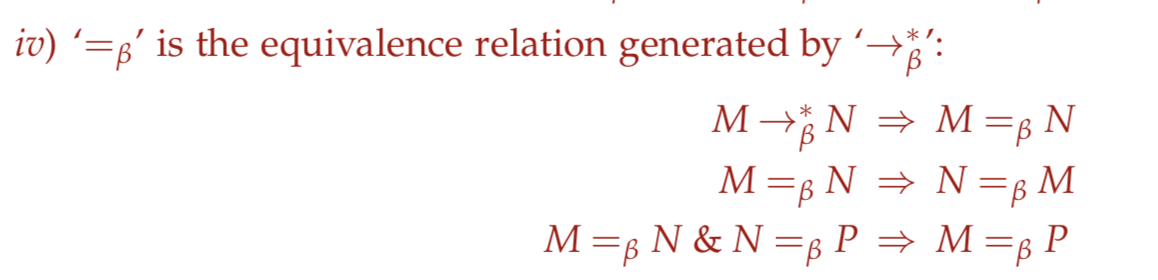
unification 1

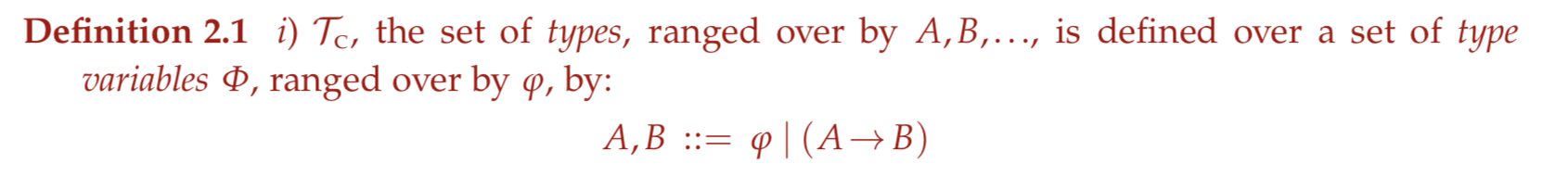
(a)

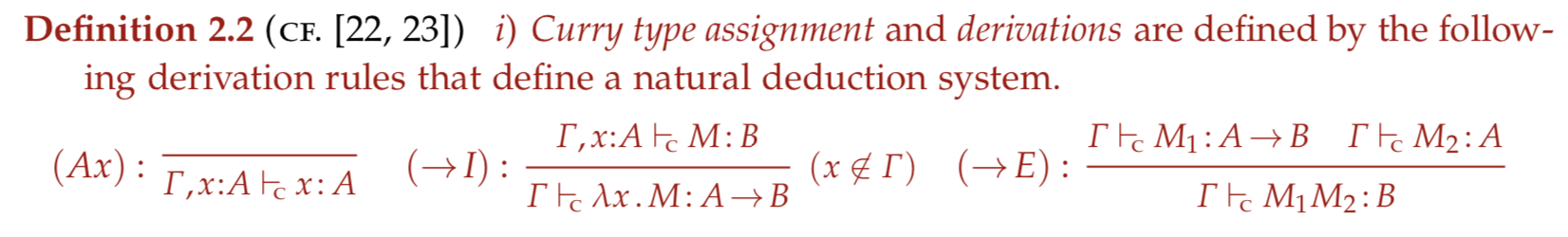
Omitted, see notes

Definition 1.1, 1.4, 2.1, 2.2









(b) See Exercise 2.23

pp S = < \varnothing; (1 -> 2 -> 3) -> (1 -> 2) -> 1 -> 3 >

pp K = < \varnothing; 1 -> 2 -> 1>

pp I = < \varnothing; 1 -> 1>

pp S K I = < \varnothing, 7 -> 7 >

-> pp S K = < \varnothing; (4 -> 2) -> 4 -> 4 >

-> pp S = < \varnothing; (1 -> 2 -> 3) -> (1 -> 2) -> 1 -> 3 >

-> pp K = < \varnothing; 4 -> 5 -> 4>

-> unify ((1 -> 2 -> 3) -> (1 -> 2) -> 1 -> 3) ((4 -> 5 -> 4) -> 6)

= 1→ 4, 3→ 4, 6→ (4 -> 2) -> 4 -> 4

-> unifyContext \varnothing \varnothing

= id

-> pp I = < \varnothing; 7 -> 7>

-> unify ((4 -> 2) -> 4 -> 4) ((7 -> 7) -> 8)

= 4→ 7, 2→7, 8→ 7 -> 7

-> unifyContext \varnothing \varnothing

= id

pp (\xy. xy) I = < \varnothing, 3-> 3 >

-> pp \xy.xy = < \varnothing, (1 -> 2) -> 1 -> 2 >

-> pp I = < \varnothing, 3 -> 3 >

-> unify ((1 -> 2) -> 1 -> 2) ((3 -> 3) -> 4)

= 1 → 3, 2 → 3, 4→ 3 -> 3

-> unifyContext \varnothing \varnothing

= id

You can then conclude that the sets of assignable types are the same for both of them.

(c)

Curry’s type system only assigns types to a subset of lambda-terms, namely strongly normalisable ones. This makes recursions impossible. Recursive functions are not always normalisable so they cannot be typed by Curry’s type system. Extending the type for recursion is achievable in multiple means, one of them being adding typing rules of the fixed-point constructor Y

\Gamma |- Y : (A -> A) -> A

(d)

let Y be fixed point constructor

To show YM =b M(YM)

YM

-> (λxy. xyx) (λyx. y(xyx)) M

-> (λy. (\yx. y(xyx)) y (λyx. y(xyx))) M

-> (λyx. y(xyx)) M (λyx. y(xyx))

-> (λx. M(xMx)) (λyx. y(xyx))

-> M ((λyx. y(xyx)) M (λyx. y(xyx)))

M(YM)

-> M ((λxy. xyx) (λyx. y(xyx)) M)

-> M ((λy. (λyx. y(xyx)) y (λyx. y(xyx))) M)

-> M ((λyx. y(xyx)) M (λyx. y(xyx)))

YM =b M ((λyx. y(xyx)) M (λyx. y(xyx))) =b M(YM)

Not typable. In \xy. xyx, x has shape X -> Y -> Z and X, this is impossible to unify

(e) See Exercise 2.24

~~SII(SII) is self application, unable to type~~

Cannot construct type SII due to infinite type

\gz. g(z (\xy.x)) (z (\x.x))

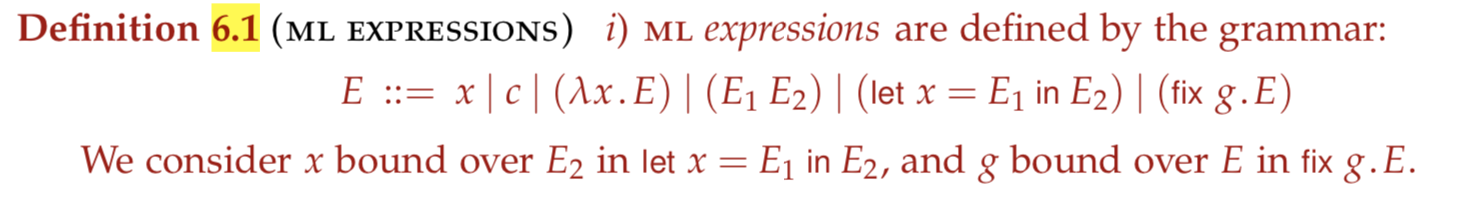
z is applied with both 1 -> 2 -> 1 and 1 -> 1

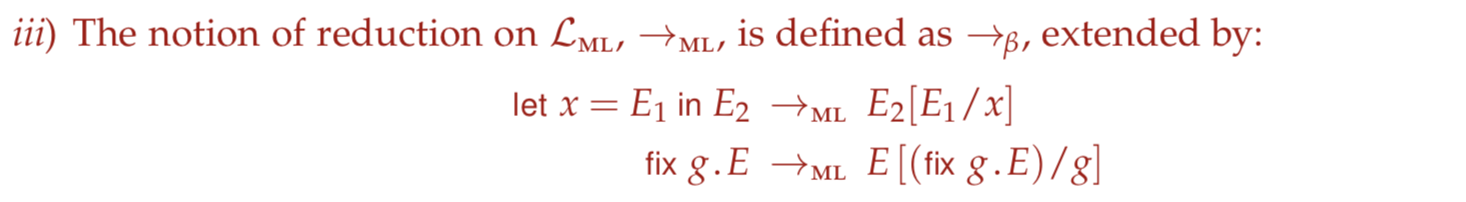
this is impossible to unify, unable to type

2

(a)

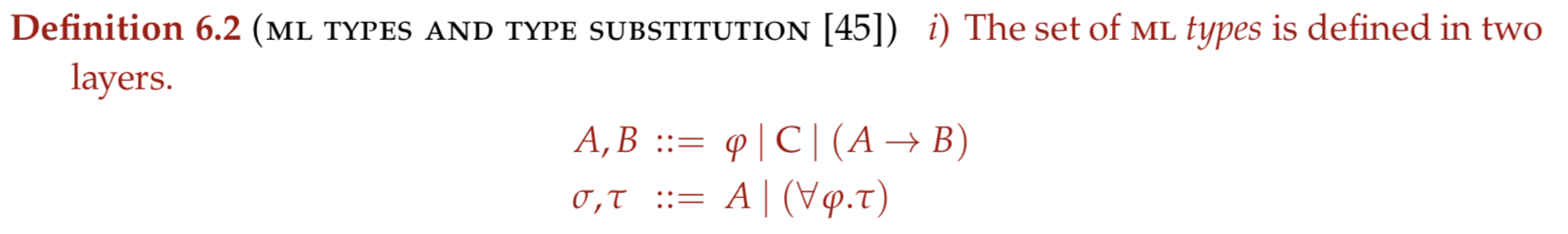
Definition 6.1

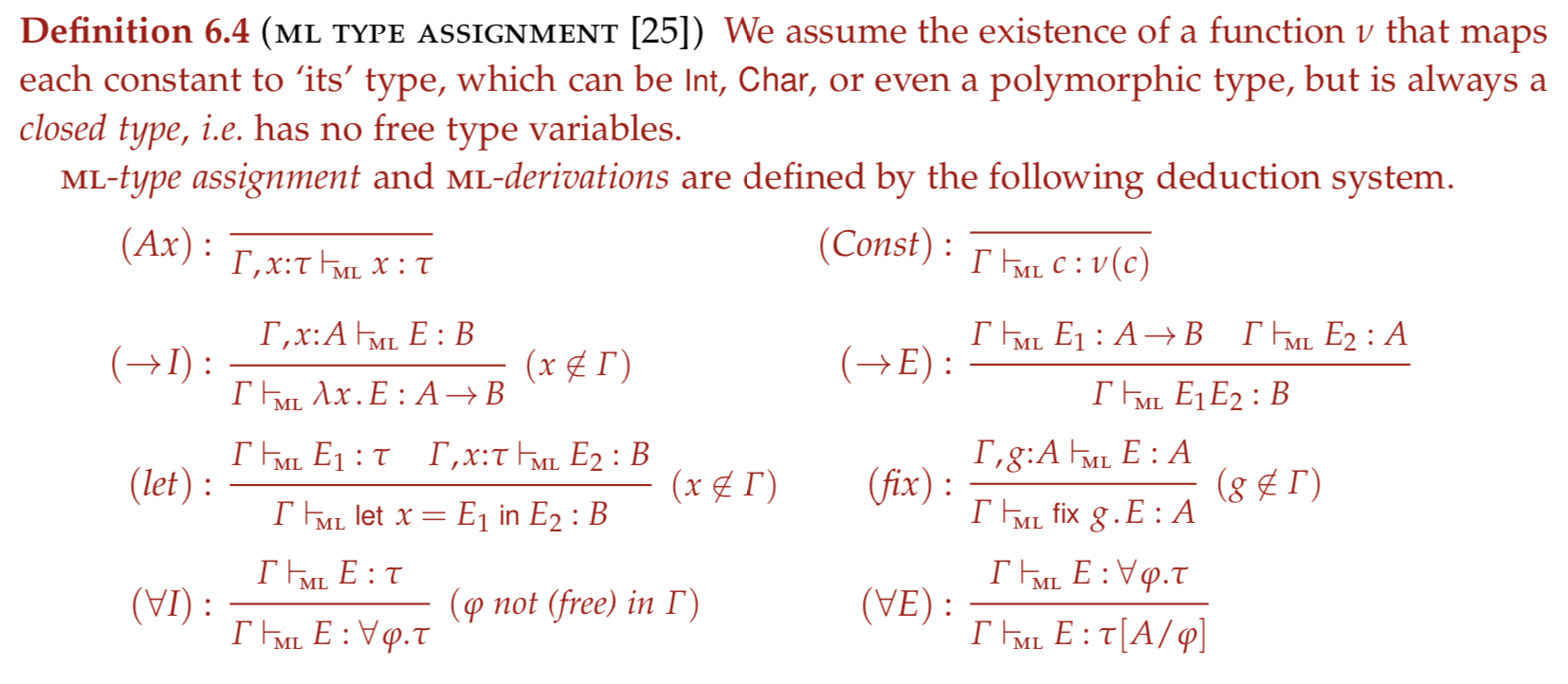




(b)

Definition 6.2 and 6.4





(c)

(i) A -> A

(ii) untypeable

(iii) A -> B

(iv) untypeable

(d) Polymorphism allows for functions to take in inputs of different types.

ML has explicit polymorphic types (\forall stuff)

You can calculate the principal type for a term and then take a fresh instance of this type for each occurrence (page 28)

(e)

(fix f. \xy. Cons x (f y (+ x y))) 0 1

This has type List Int

f has type Int -> Int -> List Int

x has type Int, y has type Int, (+ x y) has type Int

f y (+ x y) has type List Int

\*Note: Use \forall E for Cons to get an instance with \phi = Int

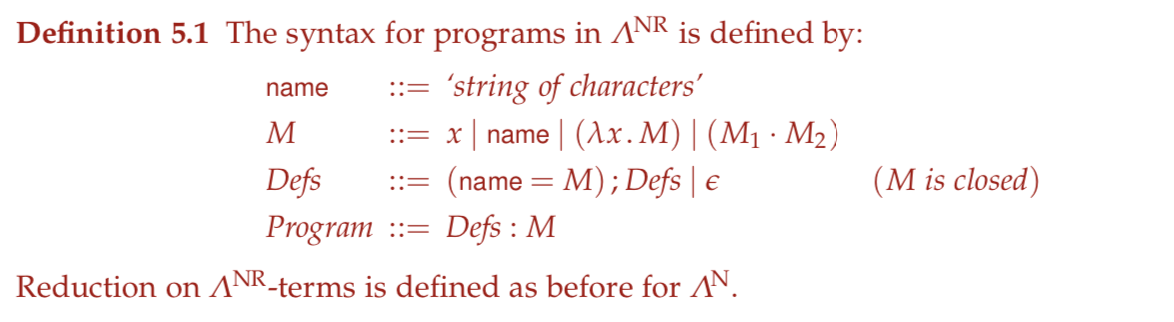
Cons x (f y (+ x y)) has type Int

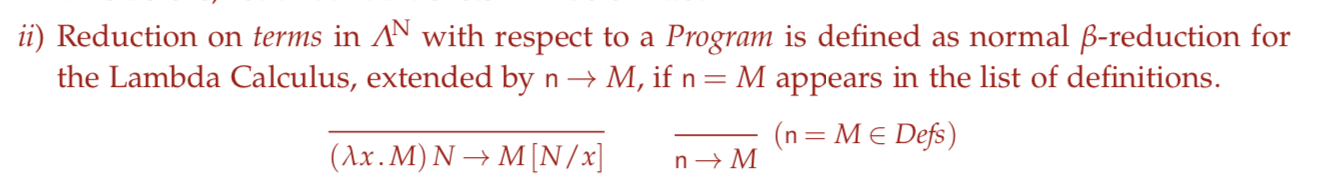
Everything works, drawing a derivation gives me cancer

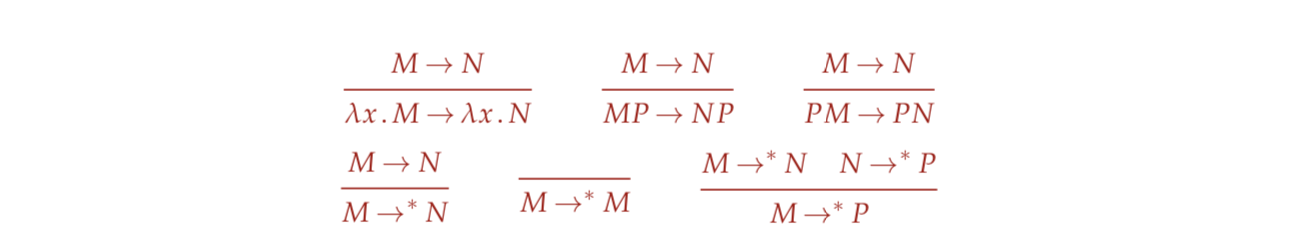
3

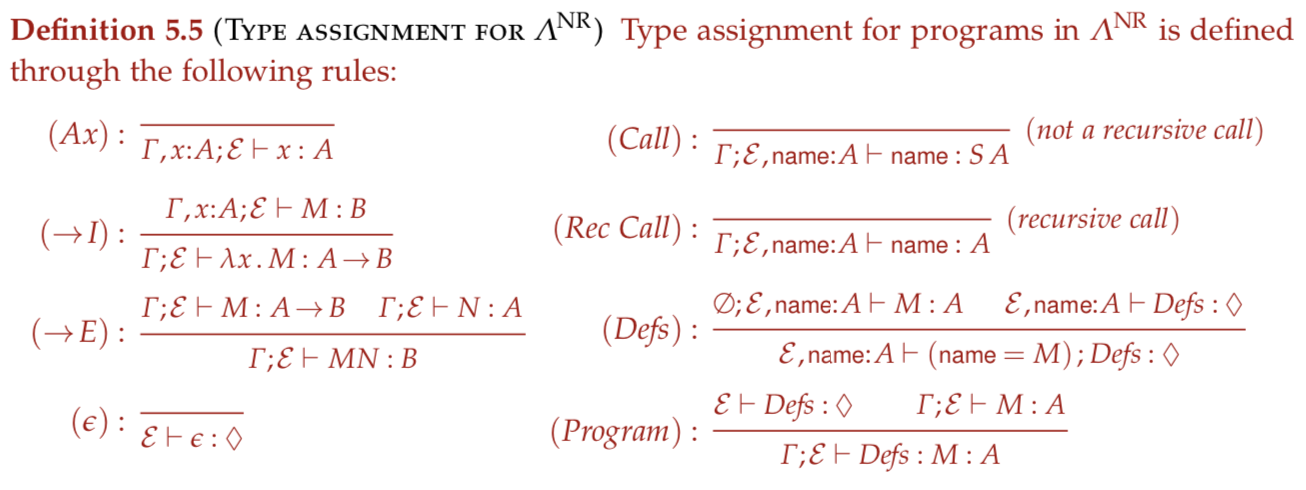
(a)(b)(c) refer to notes

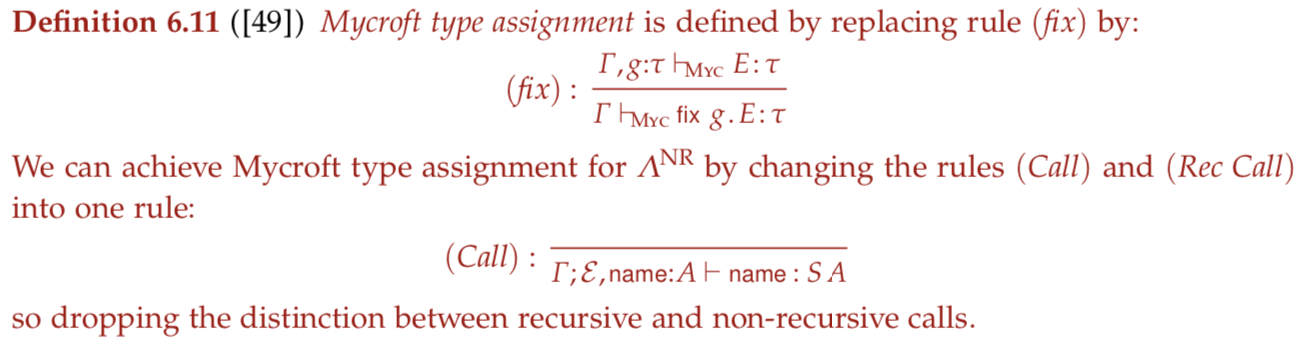
Definition 5.1, 5.5, 6.11, Section 6.4



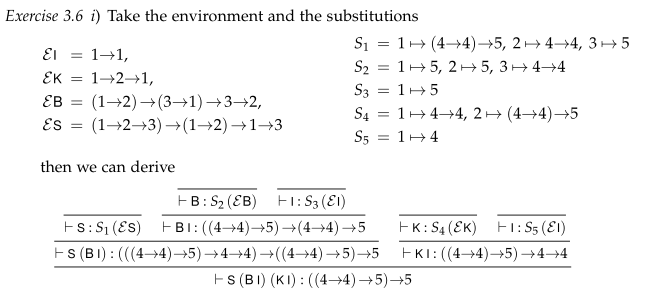








(d) type in the end is ((2->2)->1)->1. Derivation is omitted



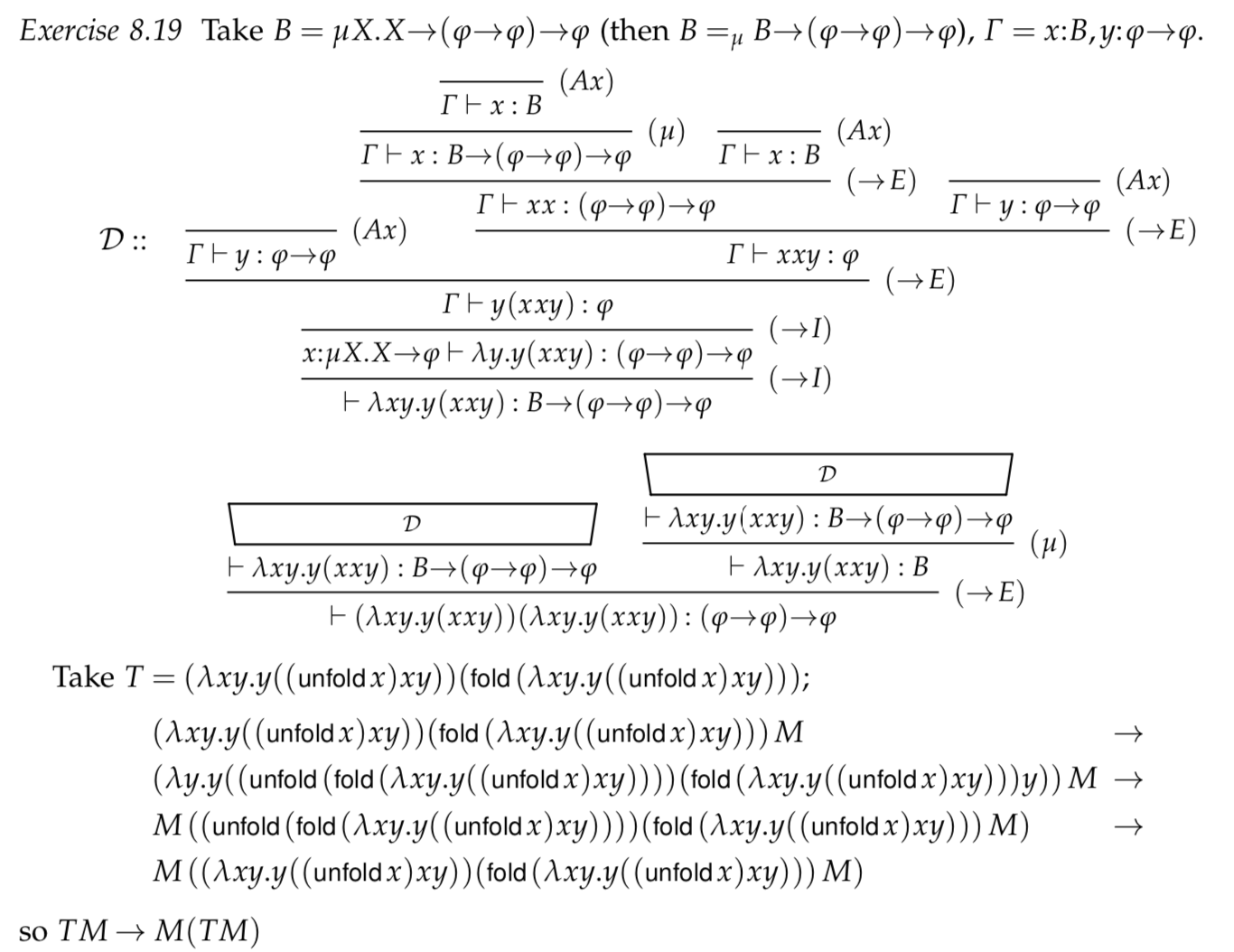
S (B I) (K I) reduces to \z z (\x.x), which has type ((2->2)->1)->1

4

Whole question is in notes

(a) (b) De1`finition 8.1, 8.2, 8.3, 8.4, 8.5, 8.8

(c)



(d) 