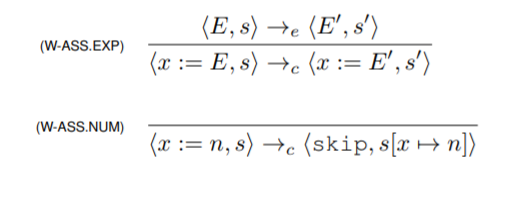
3)

a)

i)



ii)

<y := y + x, s>

-> <y := 1 + x, s>

-> <y := 1 + 3, s>

-> <y := 4, s>

-> <skip, s[y ↦ 4]>

-------------------------- s(y) = 1

<y, s> → <1, s>

---------------------------

<y + x, s> → <1 + x, s>

---------------------------

<y := y + x, s> → <y := 1 + x, s>

b) i)

First prove that <E, s> →e <E’, s’> => s = s’ (trivial structural induction by small-step semantics and IH).

---

Lemma:

<E, s> →e <E’, s’> => s = s’

---

We prove this by induction on k in the following statement:

<E, s> →k <n, s’>

Base Case:

<E, s> →e0 <n, s’>

Vacuously true.

Inductive Case:

Assume I.H: <E, s> →k <n, s’> => s = s’

To Show:

<E, s> →ek+1 <n, s’> => s = s’

Assume <E, s> →ek+1 <n, s’>.

Then <E, s> →ek+1 <n, s’> = <E, s> →e <E’, s’’> →ek <n, s’>

So by lemma s = s’’, and then by I.H s’’ = s’. Hence s = s’.

ii) As there is only one way to evaluate an expression E (i.e. there is no choice on which term to evaluate first), thus determinacy holds. Therefore E1 = E2 holds. By the results shown in b.i, evaluating expression E has no effect on state. Therefore s = s1 = s2 holds.

Inductively:

**See (Lecture 3 page 24) for a very similar proof.**

c)

i) It does not hold for b.i as the command C in the expression do C return E can be an assignment (from a.i) and thus change the state, thereby allowing a contradiction to occur with the statement of b.i. which is that evaluating expression E does not change state.

However, as determinacy holds even for the extended set of expressions (i.e. there is no choice in which to evaluate first), then E1 = E2 still holds. We can then prove s1 = s2 also holds by induction on the definition of E.

ii)

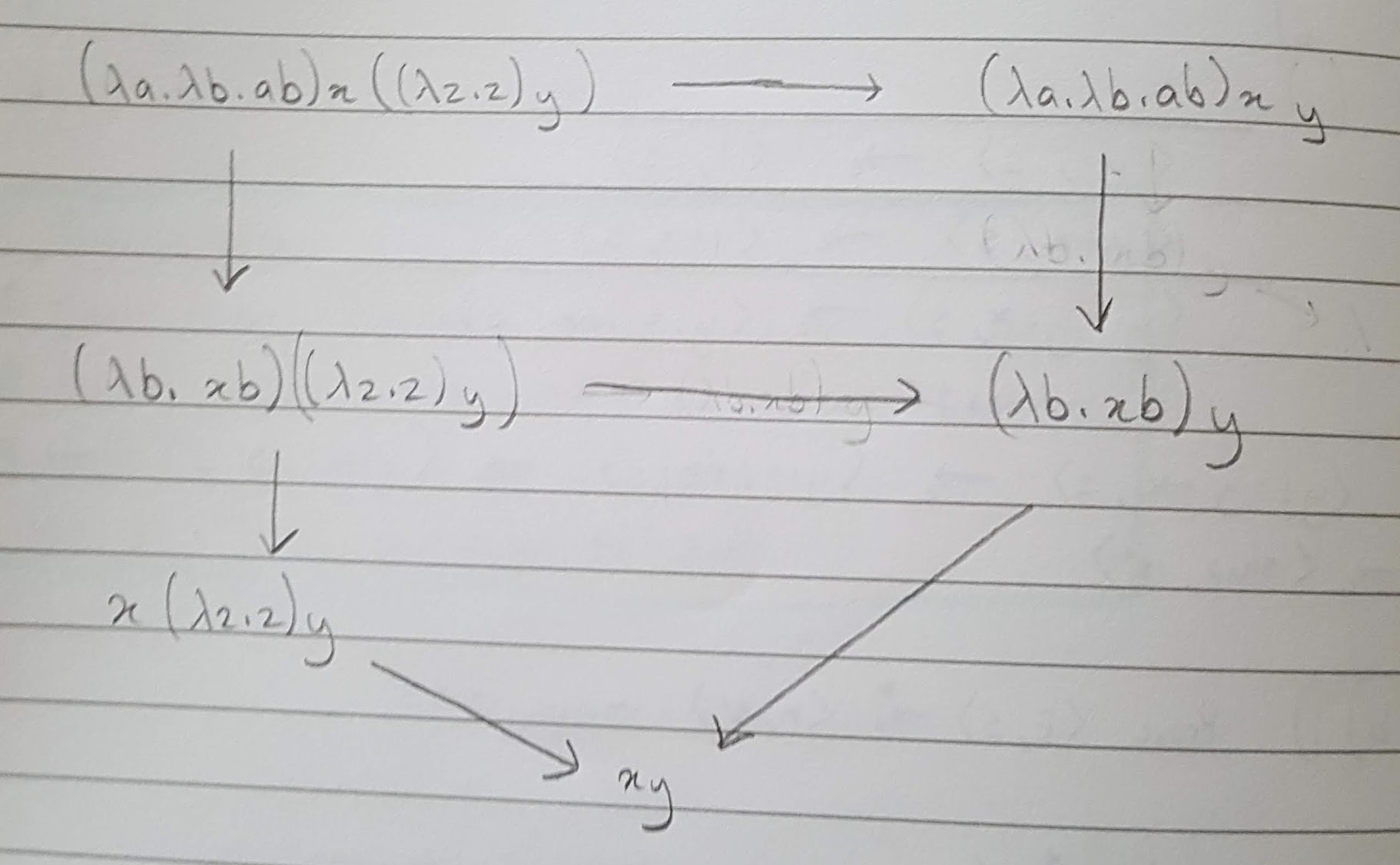
From notes: ( do x := (x + 1) return x) + ( do x := (x × 2) return x)

But this is assuming multiplication is defined, which idk if it is in this context. To get the same thing with just + do it like: ( do x := (x + 1) return x) + ( do x := (x + x) return x)

Simpler example: <y + (do y:=2 return 3), s[y ↦1]>

4)

a) "When applying reduction rules to terms in some typed variants of the lambda calculus, the ordering in which the reductions are chosen does not make a difference to the eventual result" - [Wikipedia](https://www.wikiwand.com/en/Church%E2%80%93Rosser_theorem)



b)

succ n = (λn.λf.λx f ( n f x )) n

= λf.λx. f ( n f x )

= λf.λx. f (( λf. λx. fn(x)) f x )

= λf.λx. f (( λx. fn(x)) x)

= λf.λx. f ( fn(x))

= λf.λx. fn+1(x)

= n + 1

c)

Plus m n = (λm.λn.λf.λx. m f ( n f x ) ) m n

= (λn.λf.λx. m f ( n f x ) ) n

= λf.λx. m f ( n f x )

= λf.λx. m f ( (λf. λx. fn(x)) f x )

= λf.λx. m f ( (λx. fn(x)) x )

= λf.λx. m f ( fn(x) )

= λf.λx. (λf. λx. fm(x)) f ( fn(x) )

= λf.λx. (λx. fm(x)) ( fn(x) )

= λf.λx. (fm(fn(x)))

= λf.λx. (fm+n)

= m + n

d) APlus m n = (λm.λn. m Succ n) m n

= m Succ n

= (λf. λx. fm(x)) Succ n

= Succm n

= Succm-1 Succ n

= Succm-1 n + 1 (By result shown in b)

= n + m (By repeated application of Succ) deez nuts

APlus m n reaches the same form as Plus m n, thus it can be said that they're beta equivalent.

e)

Mult = λm.λn. m (Plus n) 0

Plus n to 0, m times