

Exercise 1 (Example of Type Inference for let)

Consider the typing problem

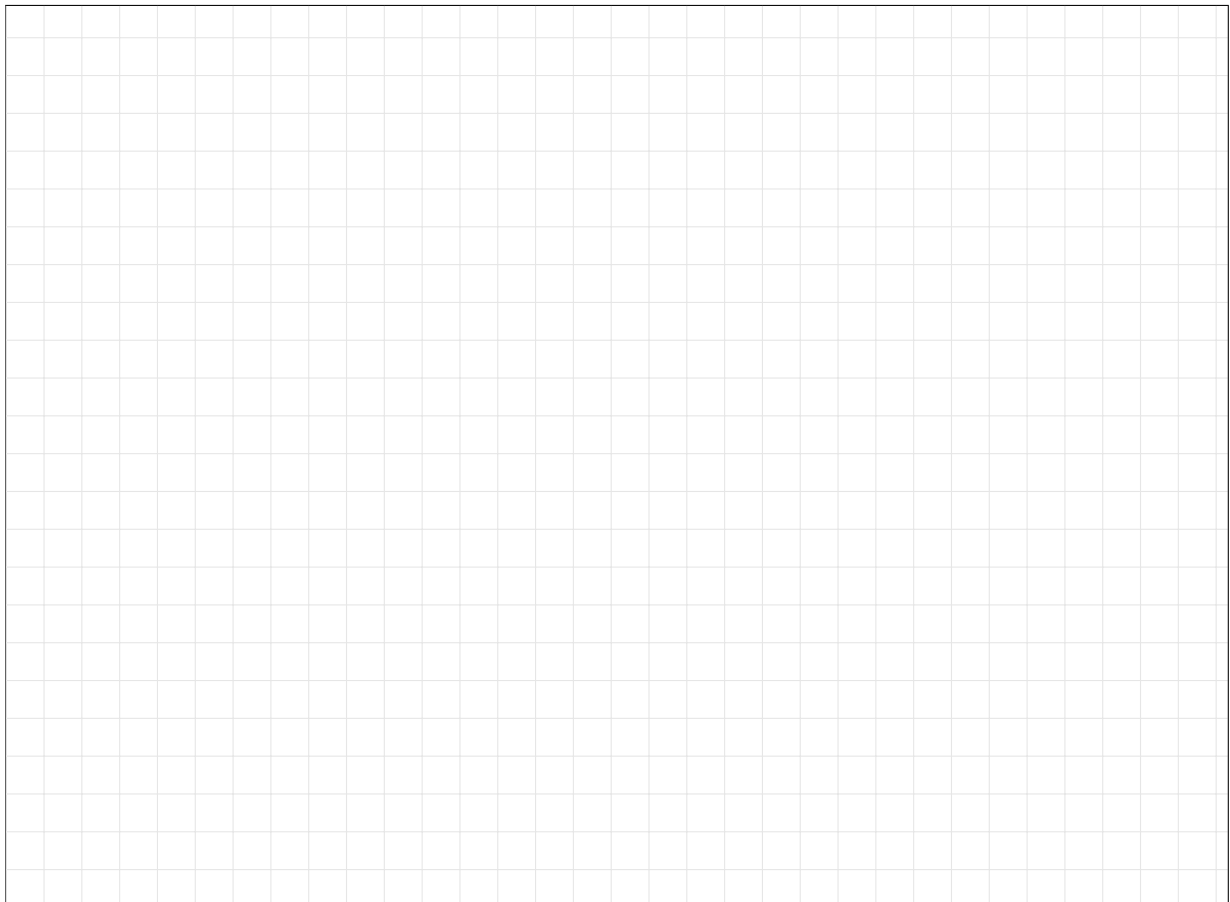
$$x : \alpha \vdash \text{let } y = \lambda z. z \ x \text{ in } y \ (\lambda v. x) : ?\tau$$

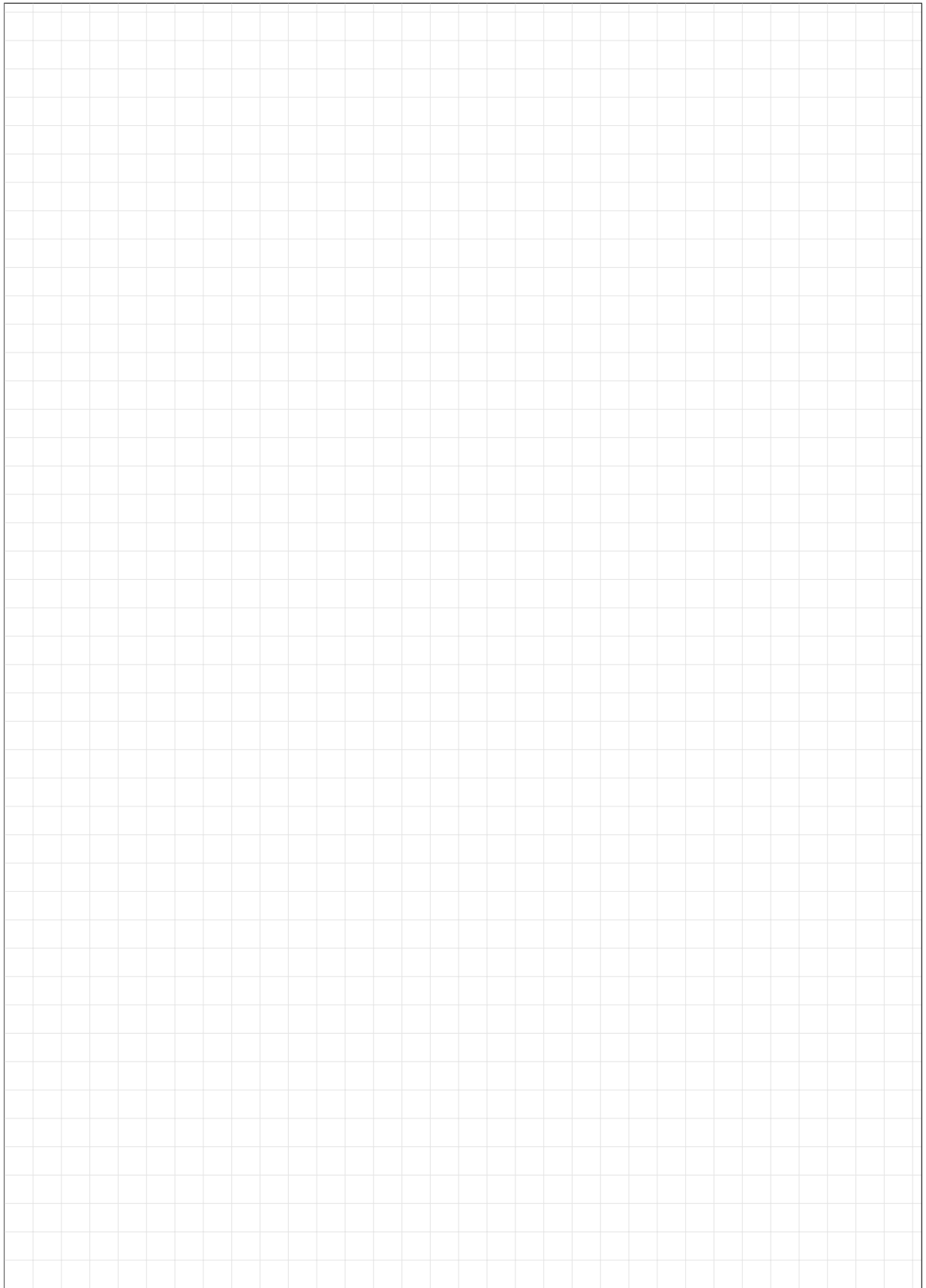
where α is a type variable.

- a) Find the most general type schema σ with $x : \alpha \vdash \lambda z. z \ x : \sigma$ and draw a type derivation tree.
- b) Draw the type derivation tree for

$$y : \sigma, x : \alpha \vdash y \ (\lambda v. x) : ?\tau$$

with the correct type for $?\tau$.



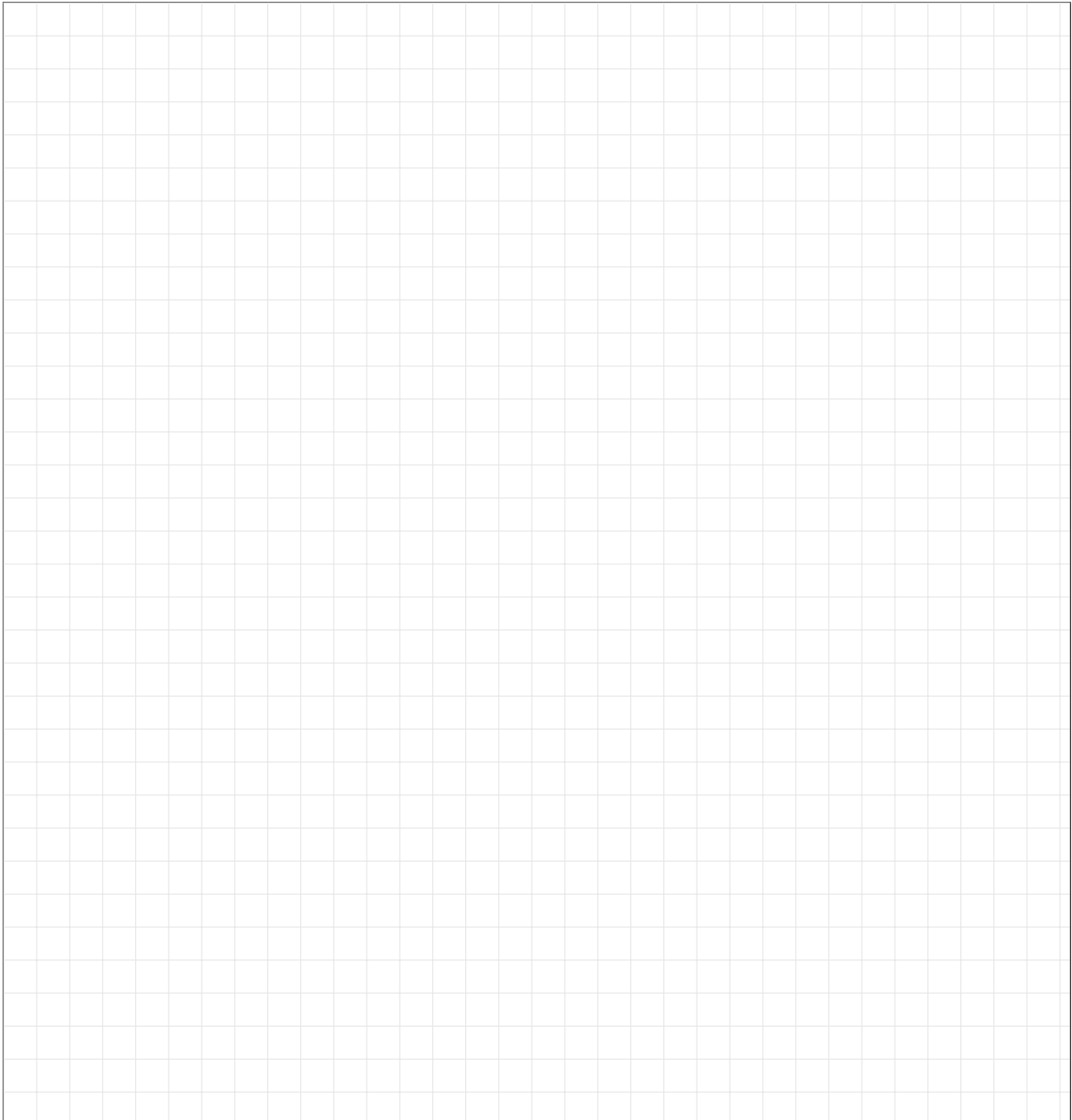


Exercise 2 (Recursive let)

Recursive `let` expressions are one way (besides Y -combinators) to add recursion to λ^\rightarrow .

$$t ::= x \mid (t_1 t_2) \mid (\lambda x. t) \mid \mathbf{letrec} \ x = t_1 \ \mathbf{in} \ t_2$$

- a) Modify the standard typing rule for `let` to create a suitable rule for `letrec`.
- b) Considering *type inference*, what is the problematic property of this rule compared to the rule for `let`?



Exercise 3 (Type Inference in Haskell (2))

Extend the implementation of the type inference algorithm from the last exercise with `let` and `letrec` constructs.

You can find a template [here](#).

$$\frac{\frac{\frac{\vdots}{\Gamma \vdash t_1 : \tau}}{\Gamma \vdash t_1 : \forall \alpha_n. \tau} \forall\text{Intro}}{\Gamma \vdash t_1 : \forall \alpha_1, \dots, \alpha_n. \tau} \forall\text{Intro}$$