Exercise 1 (Recursive let)

Recursive let expressions are one way (besides $Y$-combinators) to add recursion to $\lambda\to$.

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

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?

c) Give a derivation tree for the following statement, and so determine the type $\tau$:

$$[] \vdash \text{letrec } x = \lambda y. \ x \ (x \ y) \ \text{in } x \ : \ \tau$$

Exercise 2 (Type Inference in Haskell (2))

In this exercise, we will extend the implementation of the type inference algorithm from last exercise for the let and letrec constructs.

a) Have a look at the template on the website, which adds let and letrec to the term language.

b) Extend the type inference algorithm for let and letrec.

Homework 3 (Fixed-point combinator)

Let

$$\$$ = $\lambda abcdefghijklmnopqrstuvwxyz.r.(this\isa\fixedpointcombinator)$

and

$$\$$ = $\$$

Show that $\$$ is a fixed-point combinator.

Homework 4 (let-Polymorphism)

Give a derivation tree for the following statement, and so determine the type $\tau$:

$$[z : \tau_0] \vdash \text{let } x = \lambda yz. \ z \ y \ \text{in } x \ (x \ z) : \tau$$
Homework 5 (Type Inference with Type Constructors)

We generalize \( \to \) to type constructors. With type constructors, types are either elementary, a type variable or, constructed as \( \Pi \, \tau_1 \ldots \tau_n \) where \( \Pi \) is a type constructor. Now \( \to \) is just a type constructor that takes two arguments. Your task is to extend the Haskell inference algorithm towards type constructors.

a) Extend the type language with type constructors. Type constructors should take lists of type arguments.

b) Extend the type inference algorithm for type constructors. To specify the set of valid type constructors, we will just start with a non-empty environment that will pre-specify the type of some free variables that then act as data constructors.