Coursework **Due: 8pm, 30 April 2021**

Instructions Complete the given assignments in the file `Coursework.hs`, and submit this on Moodle by Friday 30 April 8pm. Make sure your file does not have **syntax errors** or **type errors**; where necessary, comment out partial solutions. Use the provided function names. You may use auxiliary functions, and you are encouraged to add your own examples and tests.

Assessment and feedback Your work will be judged primarily on the correctness of your solutions. Incorrect or partial solutions may be given partial marks if they operate correctly on certain inputs. Marking is part-automated, part-manual. You will receive individual marks, and we will publish an overall feedback document.

Plagiarism warning The assessed part of this coursework is an **individual assignment**. Collaboration is not permitted: you should not discuss your work with others, and the work you submit should be your own. Disclosing your solutions to others, and using other people's solutions in your work, both constitute plagiarism: see `http://www.bath.ac.uk/quality/documents/QA53.pdf`.

Type inference (100 marks)

Type inference is the process of finding a type for a λ-term. In this coursework we will implement a simple and transparent version of it. In the file `Coursework.hs`, you are started off with the implementation of the λ-calculus from the tutorials. The work is split up into several assignments. These do not always build on each other, or sometimes only partly, so if you get stuck, please do attempt the next assignment.

Types with variables

Types are defined by the following grammar, where $A = \{\alpha, \beta, \gamma, \dots\}$ is a set of type variables, here called **atoms**.

$$\rho, \sigma, \tau, \upsilon \ ::= \ \alpha \in A \mid \sigma \to \tau$$

You are given a data type `Type` that implements the above definition, with a `Show` instance for pretty printing. The set of **atoms** is given by the type `Atom`, a synonym for `String`. The list `atoms` can be used to generate a fresh atom when needed. There are two example types:

```
*Main> t1
a -> b
*Main> t2
(c -> d) -> e
```

Note that the type constructor `:->` is **infix**. The **patterns** for `Type` are as follows.

```
At x
x :-> y
```

The declaration `infixr 5 :->` moreover tells the compiler that `:->` is right-associative, like the arrow type of the λ-calculus, which means that:

```
r :-> s :-> t    ==    r :-> (s :-> t)
```

Assignment 1 (10 marks):

a) Complete the function `occurs` that determines if an atom occurs in a type.

b) Complete the function `findAtoms` that returns the atoms occurring in a type in an (alphabetically) **ordered** list.

```
*Main> occurs "a" t1
True
*Main> occurs "b" t2
False
*Main> findAtoms t1
["a","b"]
*Main> At "a" :-> t2 :-> t1
a -> ((c -> d) -> e) -> a -> b
*Main> findAtoms it
["a","b","c","d","e"]
```

Substitutions

Substitution for types is simpler than for terms, since there is no variable capture, and so no need for capture-avoidance. The definition is:

$$\begin{aligned} \alpha[\tau/\alpha] &= \tau \\ \beta[\tau/\alpha] &= \beta && \text{(if } \alpha \neq \beta\text{)} \\ (\rho \to \sigma)[\tau/\alpha] &= \rho[\tau/\alpha] \to \sigma[\tau/\alpha] \end{aligned}$$

A series of substitutions is abbreviated by S:

$$S = [\tau_1/\alpha_1] \ldots [\tau_n/\alpha_n]$$

The **empty** series of substitutions, which represents the substitution that does nothing, is written as ε. Then $\tau[S]$ is τ with the substitutions in S applied to it, so that $\tau[\varepsilon] = \tau$, and if S is as above,

$$\tau[S] = \tau[\tau_1/\alpha_1]\dots[\tau_n/\alpha_1]\,.$$

We will give a substitution $[\tau/\alpha]$ as a pair `(a,t)` of an `Atom a` and a `Type t`. A series of substitutions will be a list of such pairs (note that the order is reversed):

$S = [\tau_1/\alpha_1]\dots[\tau_n/\alpha_n]$ `[ (an,tn) , ... , (a1,t1) ]`

Assignment 2 (10 marks):

You are given a type `Sub` for substitutions, as a pair of an `Atom` and a `Type`, and three example substitutions `s1`, `s2`, and `s3`.

a) Complete the function `sub` that applies a substitution to a type.

b) Complete the function `subs` that applies a list of substitutions to a type, with the head of the list applied last, and the tail applied first.

```
*Main> sub s1 t1
e -> b
*Main> sub s2 t2
(c -> d) -> b -> c
*Main> sub s3 t3
a -> (a -> a) -> a -> a
*Main> subs [s1] t1
e -> b
*Main> subs [s2,s1] t1
(b -> c) -> b
*Main> subs [s3,s2,s1] t1
(b -> a -> a) -> b
*Main> subs [s1,s2,s3] t3
e -> (e -> e) -> e -> e
*Main> subs [s3,s2,s1] t3
(b -> a -> a) -> (a -> a) -> a -> a
```

Unification

To find a type for a term, we have to solve the following problem. First, an example. Suppose we have the function $N = \lambda x.\lambda y.x\,y\,y$ with type $(\alpha \to \alpha \to \beta) \to \alpha \to \beta$, and we want to supply it with the argument $M = \lambda z.\lambda w.z$ with the type $\gamma \to \delta \to \gamma$. To make this possible, informally we can observe that N and M can **also** be given the types:

$$\begin{aligned} N &: (\alpha \to \alpha \to \alpha) \to \alpha \to \alpha \\ M &: \alpha \to \alpha \to \alpha \end{aligned}$$

These match: N can be applied to M, with the type $N\,M : \alpha \to \alpha$. To obtain these matching types from the original ones in a more formal way, we use the series of substitutions $S = [\alpha/\beta][\alpha/\gamma][\alpha/\delta]$, which replaces each occurrence of β, γ, and δ with α. When used on both original types, we get the desired types for N and M:

$$\begin{aligned} ((\alpha \to \alpha \to \beta) \to \alpha \to \beta)[S] &= (\alpha \to \alpha \to \alpha) \to \alpha \to \alpha \\ (\gamma \to \delta \to \gamma)[S] &= \alpha \to \alpha \to \alpha \end{aligned}$$

This is the general problem: if we have $N : \sigma \to \tau$ and $M : \rho$, to give a type to $N\,M$ we need to find substitutions S such that $\sigma[S] = \rho[S]$, and then we will have $N\,M : \tau[S]$. (The reason that we can use substitution to make types equal, is that if a term N has type τ, then it can also be given the more special type $\tau[S]$ for any substitutions S.)

The algorithm that finds S is called **unification**, and the substitutions S that give $\rho[S] = \sigma[S]$ are called a **unifier** of ρ and σ. Here are a few observations for how it should work.

- A unifier for a variable α and any type τ can be just the substitution $[\tau/\alpha]$, since $\alpha[\tau/\alpha] = \tau$.
- **Unless** τ contains the variable α, for example if $\tau = \beta \to \alpha$. In that case, $(\beta \to \alpha)[\beta \to \alpha/\alpha] = \beta \to \beta \to \alpha$ while $\alpha[\beta \to \alpha/\alpha] = \beta \to \alpha$, and these are not equal. Then, unification fails: α and $\beta \to \alpha$ cannot be unified.
- **Unless** τ is just α itself, since α can be unified with itself by the empty substitution series ε.
- A unifier for two types that are not variables, $\sigma_1 \to \sigma_2$ and $\tau_1 \to \tau_2$, must be a substitution series S such that $\sigma_1[S] = \tau_1[S]$ **and** $\sigma_2[S] = \tau_2[S]$.

This last case shows that the algorithm needs to find a unifier not just for a single pair of types, but for a **set** of pairs of types. The four cases above then show what steps it must take: if a pair is of the form

- α and α, discard it; otherwise

- α and τ containing α, fail to unify these terms;
- α and τ not containing α, return the unifier $[\tau/\alpha]$;
- $\sigma_1 \to \sigma_2$ and $\tau_1 \to \tau_2$, continue to unify both the pair σ_1 and τ_1 and the pair σ_2 and τ_2.

We will now formalize the unification algorithm. We will write a pair of types σ and τ that we want to unify as $\sigma \leftrightarrow \tau$, and call it a **unification pair**. The algorithm is then defined using **states** and **transitions**. A **state** is a pair (S, U) where

- S is a series of subsitutions, which is the solution so far;
- $U = \{\sigma_1 \leftrightarrow \tau_1, \dots, \sigma_n \leftrightarrow \tau_n\}$ is a set of unification pairs.

To unify σ and τ the algorithm starts from $(\varepsilon, \{\sigma \leftrightarrow \tau\})$. The **transitions** are as follows: on input $(S, \{u\} \cup U)$, if the unification pair u is of the form

a) $\alpha \leftrightarrow \alpha$: return (S, U);

b) $\alpha \leftrightarrow \tau$ or $\tau \leftrightarrow \alpha$: if α occurs in τ, **FAIL**;
   otherwise, return $(S[\tau/\alpha], U[\tau/\alpha])$

c) $(\sigma_1 \to \sigma_2) \leftrightarrow (\tau_1 \to \tau_2)$: return $(S, \{\sigma_1 \leftrightarrow \tau_1, \sigma_2 \leftrightarrow \tau_2\} \cup U)$

Here, $S[\tau/\alpha]$ is the series of substitution S with $[\tau/\alpha]$ added at the end, and $U[\tau/\alpha]$ is the set of unification pairs U with $[\tau/\alpha]$ applied to every type in every pair. The unification process is complete when a state $(S, \varnothing)$ is reached, with an empty set of unification pairs. The algorithm then returns S as the unifier.

Assignment 3 (30 marks):

To implement and test the unification algorithm, you are given a type `Upair` for unification pairs, a type `State` for the internal state of the algorithm, example unification pairs `u1` through `u4`, and example state `st1`.

a) Complete the function `sub_u` that applies a substitution $[\tau/\alpha]$ to a list of unification pairs U as $U[\tau/\alpha]$.

b) Complete the function `step` that carries out a single transition of the unification algorithm as described above. For the case of a **FAIL**, you may throw an error.

c) Complete the function `unify` that does the following:

   - Given a list of unification pairs U, start the algorithm with the state (ε, U)

- If the current state is $(S, \varnothing)$, return S.
- Otherwise, apply a transition and try again.

```
*Main> sub_u s1 [u1,u2]
[(e -> b,c),(e -> e,e -> c)]
*Main> sub_u s3 [u4]
[(((a -> a) -> d) -> e,a -> (a -> a) -> a -> a)]
*Main> st1           -- highlighting the next unification pair
([],[(a -> b,c),(a -> a,a -> c)])
*Main> step it                                -- case b)
([("c",a -> b)],[(a -> a,a -> a -> b)])
*Main> step it                                -- case c)
([("c",a -> b)],[(a,a),(a,a -> b)])
*Main> step it                                -- case a)
([("c",a -> b)],[(a,a -> b)])
*Main> step it                                -- case b), FAIL
*** Exception: Step: atom a occurs in a -> b
*Main> unify [u3,u4]
[("e",c -> c),("b",e),("a",c -> d)]
*Main> let s = it
*Main> subs s t1
(c -> d) -> c -> c
*Main> subs s t2
(c -> d) -> c -> c
*Main> subs s t3
(c -> d) -> c -> c
```

Derivations

To give a term a type is done by a **typing derivation**. We need the following notions:

- A **context** $\Gamma = x_1 : \tau_1, \dots, x_n : \tau_n$ is a sequence of variables x_1 through x_n each with an assigned type τ_1 through τ_n. The order of variables is not important, and it is assumed that variables don't occur more than once.
- A **judgement** $\Gamma \vdash N : \tau$ assigns a term N the type τ in the context Γ.
- A **derivation** is a proof that a judgement holds, and is built from the following typing

rules, called **axiom**, **abstraction**, and **application**

$$\dfrac{}{\Gamma, x : \tau \vdash x : \tau}\ \text{axiom} \qquad \dfrac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda x.M : \sigma \to \tau}\ \text{abstraction} \qquad \dfrac{\Gamma \vdash M : \sigma \to \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \tau}\ \text{application}$$

An example derivation for the term $(\lambda x.\, x)\, y$ is as follows.

$$\dfrac{\dfrac{\dfrac{}{y : \alpha, x : \alpha \vdash x : \alpha}}{y : \alpha \vdash \lambda x.\, x : \alpha \to \alpha} \qquad \dfrac{}{y : \alpha \vdash y : \alpha}}{y : \alpha \vdash (\lambda x.\, x)\, y : \alpha}$$

Assignment 4 (20 marks):

a) Complete the following types:

- The type synonym `Context` for contexts as lists of pairs of a variable and a type. (In your file, `Context` is set to the unit type `()` as a default; you should replace this with your own type.)
- The type synonym `Judgement` for judgements as a triple of a context, a term, and a type.
- The data type `Derivation` for derivations, with the three constructors `Axiom`, `Abstraction`, and `Application` for each of the typing rules. Each constructor should take as arguments: a `Judgement` for the conclusion of the rule, and for each premise of the typing rule a `Derivation`.

Un-comment the test derivations `d1` and `d2` to check your definition. If the file does not typecheck, inspect `d1` and `d2` to find the correct type definitions.

Un-comment also the `Show` instance for `Derivation`, under the header "Typesetting derivations" in your `Coursework.hs` file, for pretty derivations. Evaluate `d1` and `d2` to see what it does.

b) Complete the function `conclusion` that extracts the concluding `Judgement` from a derivation.

c) Complete the three functions `subs_ctx`, `subs_jdg`, and `subs_der` that apply a list of substitutions to every `Type` in respectively a `Context`, a `Judgement`, and a `Derivation`. Like your earlier function `subs` (which you may use here) the substitution at the head of the list should be applied last.

```
*Main> d1

 --------------------
 x: a , y: a |- x : a
----------------------  -------------
y: a |- \x. x : a -> a  y: a |- y : a
-------------------------------------
        y: a |- (\x. x) y : a

*Main> conclusion d2
([("y",b)],(\x. x y) (\z. z),a)

*Main> subs_jdg [s2,s1] it
([("y",b)],(\x. x y) (\z. z),b -> c)

*Main> subs_jdg [("b",At "d" :-> At "e")] it
([("y",d -> e)],(\x. x y) (\z. z),(d -> e) -> c)

*Main> subs_der [s2,s1] d1

  -----------------------------------
  x: b -> c , y: b -> c |- x : b -> c
---------------------------------------  -----------------------
y: b -> c |- \x. x : (b -> c) -> b -> c  y: b -> c |- y : b -> c
----------------------------------------------------------------
                y: b -> c |- (\x. x) y : b -> c
```

Type inference

In this final part we will construct a type derivation for a λ-term, if it can be typed. The process will be as follows:

- First, build an incomplete derivation where every type is an atom, following the incomplete typing rules below. An example of such an incomplete derivation is d2 .
- From this derivation, extract a set of unification pairs U , one for each rule, according to the table below.
- From this set U construct a unifier S , if one exists, using the algorithm from the section **Unification**.

- Apply the substitutions in S to the incomplete derivation, using the algorithms from the previous section, **Derivations**.

Incomplete rules	**Unification pairs**
$\dfrac{}{\Gamma, x:\alpha \vdash x:\beta}$	$\alpha \leftrightarrow \beta$
$\dfrac{\Gamma, x:\alpha \vdash M:\beta}{\Gamma \vdash \lambda x.M:\gamma}$	$\gamma \leftrightarrow \alpha \to \beta$
$\dfrac{\Gamma \vdash M:\gamma \qquad \Gamma \vdash N:\alpha}{\Gamma \vdash MN:\beta}$	$\gamma \leftrightarrow \alpha \to \beta$

The result of this process, if it is successful, is a derivation; moreover, it is guaranteed to find a derivation if one exists.

Assignment 5 (30 marks):

a) Complete the function `derive0` that creates an incomplete derivation from a λ-term where all types are 'empty', as `At ""`. To manage contexts, use the auxiliary function `aux` to build a derivation from a `Judgement` instead of a term. Make sure the contexts hold the correct variables. In the abstraction case, for a term $\lambda x.N$, if x already occurs in the context, replace the old occurrence instead of adding a second x (the new x does not have to be in the same place as the old one). Initially, the context should hold the **free variables** of the input term.

b) Complete the function `derive1` that creates an incomplete derivation from a λ-term where all types are atoms, following the rules above. Use your `derive0` as an example to expand on (not as an auxiliary function to call). Make sure that all atoms you introduce are fresh, using the list `atoms` as a supply. Initially, assign an atom from `atoms` to the term itself and to each free variable in the context, and provide the remaining atoms to `aux`. To provide distinct atoms to the two premises of an application, you can split a stream of atoms in two by selecting the **even**-positioned atoms for one stream, and the **odd**-positioned ones for the other:

$$\alpha, \beta, \gamma, \delta, \varepsilon, \ldots \quad \mapsto \quad \begin{array}{l} \alpha, \gamma, \varepsilon, \ldots \\ \beta, \delta, \ldots \end{array}$$

c) Complete the function `upairs` that extracts the type unification pairs from an incomplete derivation. You may assume that the derivation is generated by `derive1`, and follows the incomplete rules above. (That is, you don't have to check the entire context from one rule to another, only the type of the term and of the abstracted variable in an abstraction rule.)

d) Complete the function `derive` that takes a term and produces a type derivation for it, if one exists. (If none exists, you may throw an error.) Do this by generating an incomplete derivation with `derive1`, extracting the unification pairs with `upairs`, unifying them with `unify`, and applying the resulting substitutions to the incomplete derivation with `subs_der`.

Note that in your solutions, you do not have to have the same type variables, as long as they are the same up to renaming.

```
*Main> derive0 n1


-----------------
x:   , y:   |- x :
-----------------  -----------
 y:   |- \x. x :     y:   |- y :
 -----------------------------
      y:   |- (\x. x) y :

*Main> derive1 n1        -- You may have different atoms,
                         -- but they must all be distinct

--------------------
x: f , y: b |- x : h
--------------------  -------------
 y: b |- \x. x : c     y: b |- y : d
 ----------------------------------
       y: b |- (\x. x) y : a

*Main> upairs it         -- You may have a different order
[(c,d -> a),(c,f -> h),(h,f),(d,b)]

*Main> upairs d2         -- You may have a different order
[(c,h -> a),(c,d -> e),(f,g -> e),(f,d),(g,b),(h,i -> j),(j,i)]

*Main> derive n1         -- You may have different atoms


 --------------------
 x: b , y: b |- x : b
----------------------  -------------
y: b |- \x. x : b -> b  y: b |- y : b
-------------------------------------
        y: b |- (\x. x) y : b
```