1 Type inference

For each of the terms below, specify its principal type scheme in the Hindley-Milner system (if it has one) for STLC with booleans, or otherwise say that the term cannot be typed.

As a reminder, the principal type scheme for a term is the most general type scheme that can be used as this term's type. For example, the principal type scheme of the identity function λx.x is $\forall a.a \to a$.

1. λ`p.  `λ`q.  p`
   $\forall a.\forall b.a \to b \to a$
2. λ`p.  (p true)`
   $\forall a.(\texttt{Bool} \to a) \to a$
3. λ`p.  `λ`q.  p q`
   $\forall a.\forall b.(a \to b) \to a \to b$
4. λ`p.  p p`
   Cannot be typed
5. λ`p.  p (p true)`
   $(\texttt{Bool} \to \texttt{Bool}) \to \texttt{Bool}$
6. λ`p.  `λ`q.  `λ`r.  p q true`
   $\forall a.\forall b.\forall c.(a \to \texttt{Bool} \to b) \to a \to c \to b$
7. λ`p.  `λ`q.  p (q (`λ`r.  r))`
   $\forall a.\forall b.\forall c.(b \to c) \to ((a \to a) \to b) \to c$
8. λ`p.  `λ`q.  p (q true) (q (`λ`r.  r))`
   Cannot be typed

2 Subtyping

In this problem, we study some properties of *algorithmic subtyping* for function types and a bottom type. Recall that, compared to declarative subtyping, algorithmic subtyping removes explicit typing rules for reflexivity (S-REFL) and transitivity (S-TRANS), but makes them provable from the algorithmic subtyping rules. The algorithmic subtyping rules we consider in this problem are presented below:

$$\texttt{Bot} <: \texttt{T} \tag{S-Bot}$$

$$\frac{\texttt{T}_1 <: \texttt{S}_1 \qquad \texttt{S}_2 <: \texttt{T}_2}{\texttt{S}_1 \to \texttt{S}_2 <: \texttt{T}_1 \to \texttt{T}_2} \tag{S-Arrow}$$

where the types are defined as follows:

$$\texttt{T} ::= \texttt{Bot} \quad | \quad \texttt{T} \to \texttt{T}$$

We define *reflexivity* and *transitivity* of subtyping as follows:

1. Reflexivity: for any type $\texttt{S}$, $\texttt{S} <: \texttt{S}$.
2. Transitivity: for all types $\texttt{S}, \texttt{U}, \texttt{T}$, if $\texttt{S} <: \texttt{U}$ and $\texttt{U} <: \texttt{T}$, then $\texttt{S} <: \texttt{T}$.

Question A

Prove *transitivity* for the algorithmic subtyping rules and types given above.

Solution:

Proof. By induction on the sum of the sizes of the subtyping derivation trees of $\texttt{S} <: \texttt{U}$ and $\texttt{U} <: \texttt{T}$.

We proceed by case analysis of the last rule used to derive $\texttt{S} <: \texttt{U}$.

Case S-BOT. We know $\texttt{S} = \texttt{Bot}$.
We apply S-BOT to get $\texttt{S} <: \texttt{T}$.

Case S-ARROW. We know $\texttt{S} = \texttt{S}_1 \to \texttt{S}_2$, $\texttt{U} = \texttt{U}_1 \to \texttt{U}_2$, $\texttt{U}_1 <: \texttt{S}_1$ and $\texttt{S}_2 <: \texttt{U}_2$.
The only rule that can be used to derive $\texttt{U} <: \texttt{T}$ is therefore S-ARROW, so we have $\texttt{T} = \texttt{T}_1 \to \texttt{T}_2$, $\texttt{T}_1 <: \texttt{U}_1$ and $\texttt{U}_2 <: \texttt{T}_2$. By the induction hypotheses, we have $\texttt{T}_1 <: \texttt{S}_1$ and $\texttt{S}_2 <: \texttt{T}_2$. Now we apply S-ARROW to get $\texttt{S} <: \texttt{T}$.

□

Historical note about when that question appeared in an exam: the proper subject of the induction was very tricky to make correctly. We accepted as correct (eligible to the full 5 points) solutions that mentioned an induction on the subtyping derivation of $\texttt{S} <: \texttt{U}$, or on both derivations. Technically, neither of those two subjects provides a sufficiently powerful induction hypothesis.

Providing the fully correct induction subject would have been eligible for a 1 point bonus, but no student found it.

Question B

If we want to add a new type `Nat` for natural numbers, what additional subtyping rule(s) will be needed to preserve *reflexivity* and *transitivity*? You do not have to prove that the properties are preserved.

Solution:

We will need a rule $\texttt{Nat} <: \texttt{Nat}$ to preserve transitivity and reflexivity.

Question C

Many new programming languages, including Scala 3, support *union types*. A value of a union type `S | T` can be either of the type `S` or type `T`. Union types are different from sum types: `Int` is a subtype of $\texttt{Int} \mid \texttt{String}$, but `Int` is not a subtype of $\texttt{Int} + \texttt{String}$. In general, both `A` and `B` are subtypes of $\texttt{A} \mid \texttt{B}$. Conversely, if `Int` and `String` share a common supertype like `Primitive`, it is possible to use an $\texttt{Int} \mid \texttt{String}$ where a `Primitive` is expected. This also generalizes to any three types `A`, `B` and `C`.

In *most* cases, union types express the same concept as sum types while being less verbose. For example, the integer `5` can take the type $\texttt{Int} \mid \texttt{String}$, while we need to write `inl 5` for it to take the sum type $\texttt{Int} + \texttt{String}$.

Extend the above algorithmic subtyping rules for union types such that *reflexivity* and *transitivity* still hold. Your rules must stay algorithmic in the sense that every type variable in the premises of a rule exists in its conclusion (S-Trans is not algorithmic by that definition, for example). You do not have to prove that the properties are preserved. You may not add reflexivity nor transitivity themselves as rules.

Solution:

$$\frac{\texttt{S} <: \texttt{T}_1}{\texttt{S} <: \texttt{T}_1 \mid \texttt{T}_2} \qquad \text{(S-Union-Left)}$$

$$\frac{\texttt{S} <: \texttt{T}_2}{\texttt{S} <: \texttt{T}_1 \mid \texttt{T}_2} \qquad \text{(S-Union-Right)}$$

$$\frac{\texttt{S}_1 <: \texttt{T} \qquad \texttt{S}_2 <: \texttt{T}}{\texttt{S}_1 \mid \texttt{S}_2 <: \texttt{T}} \qquad \text{(S-Union-Merge)}$$

Appendix: **Hindley-Milner Type System**

Type	$\mathtt{T}$	::=	$\mathtt{Bool} \mid \mathtt{T} \to \mathtt{T}$
Type Scheme	$\mathtt{S}$	::=	$\mathtt{T} \mid \forall\mathtt{X}.\mathtt{S}$

$$\frac{\mathtt{x}:\mathtt{S} \in \Gamma}{\Gamma \vdash \mathtt{x}:\mathtt{S}} \tag{T-Var}$$

$$\frac{\Gamma \vdash \mathtt{t}:\mathtt{S}_1 \qquad \mathtt{X} \notin FV(\Gamma)}{\Gamma \vdash \mathtt{t}:\forall\mathtt{X}.\mathtt{S}_1} \tag{T-TAbs}$$

$$\frac{\Gamma \vdash \mathtt{t}:\forall\mathtt{X}.\mathtt{S}_1}{\Gamma \vdash \mathtt{t}:[\mathtt{X} \mapsto \mathtt{T}_2]\mathtt{S}_1} \tag{T-TApp}$$

$$\frac{\Gamma \vdash \mathtt{t}_1:\mathtt{S}_1 \qquad \Gamma, \mathtt{x}:\mathtt{S}_1 \vdash \mathtt{t}_2:\mathtt{T}}{\Gamma \vdash \mathtt{let\ x\ =\ t}_1\ \mathtt{in\ t}_2:\mathtt{T}} \tag{T-Let}$$

$$\frac{\Gamma, \mathtt{x}:\mathtt{T}_1 \vdash \mathtt{t}_2:\mathtt{T}_2}{\Gamma \vdash \lambda\mathtt{x}.\mathtt{t}_2:\mathtt{T}_1 \to \mathtt{T}_2} \tag{T-Abs}$$

$$\frac{\Gamma \vdash \mathtt{t}_1:\mathtt{T}_2 \to \mathtt{T}_{12} \qquad \Gamma \vdash \mathtt{t}_2:\mathtt{T}_2}{\Gamma \vdash \mathtt{t}_1\ \mathtt{t}_2:\mathtt{T}_{12}} \tag{T-App}$$

$$\Gamma \vdash \mathtt{true}:\mathtt{Bool} \tag{T-True}$$

$$\Gamma \vdash \mathtt{false}:\mathtt{Bool} \tag{T-False}$$