THE TYPE INFERENCE ALGORITHM

\( \forall \)-elimination:

\[
\frac{x : \forall a_1 \ldots \forall a_n. \tau \in \Gamma}{\Gamma \vdash x : [\beta_1/a_1] \ldots [\beta_n/a_n] \tau, \beta_i \ fresh}
\]

Application:

\[
\frac{T \Gamma \vdash e_1 : \tau_1 \quad T' \tau' \Gamma \vdash e_2 : \tau_2 \quad T' \tau_1 \Upsilon \tau_2 \rightarrow \alpha \quad \alpha \ fresh}{UT' \tau' \tau \Gamma \vdash \text{apply}(e_1, e_2) : U \alpha}
\]

Function definition:

\[
\frac{T(\Gamma \cup \{x : \alpha\}) \vdash e : \tau}{\Gamma \vdash \text{letfun}(f.x.e) : T\alpha \rightarrow \tau \quad \alpha \ fresh}
\]
THE TYPE INFERENCE ALGORITHM

\( \forall \)-elimination:

\[
\frac{x : \forall a_1 \ldots \forall a_n. \tau \in \Gamma}{\Gamma \vdash x : [\beta_1/a_1] \ldots [\beta_n/a_n] \tau}, \quad \beta_i \text{ fresh}
\]

Application:

\[
\frac{T \Gamma \vdash e_1 : \tau_1 \quad T' \Gamma \vdash e_2 : \tau_2 \quad T' \tau_1 \stackrel{U}{\sim} \tau_2 \rightarrow \alpha}{UT' \Gamma \vdash \text{apply}(e_1, e_2) : U \alpha}, \quad \alpha \text{ fresh}
\]

Function definition:

\[
\frac{T(\Gamma \cup \{x : \alpha\}) \vdash e : \tau}{T \Gamma \vdash \text{letfun}(f.x.e) : T\alpha \rightarrow \tau}, \quad \alpha \text{ fresh}
\]

But wait!!! This only works for non-recursive functions!
Function definition:

\[
T(\Gamma \cup \{x : \alpha_1\} \cup \{f : \alpha_2\}) \vdash e : \tau \quad \alpha_2 \overset{U}{\sim} T\alpha_1 \rightarrow \tau
\]

\[
\text{UTT} \vdash \text{letfun}(f.x.e) : T\alpha_1 \rightarrow \tau
\]

\(\alpha_1, \alpha_2 \text{ fresh}\)
Function definition:

\[
T(\Gamma \cup \{x : \alpha_1\} \cup \{f : \alpha_2\}) \vdash e : \tau \quad \alpha_2 \sim T\alpha_1 \rightarrow \tau \\
\text{UTT} \vdash \text{letfun}(f.x.e) : T\alpha_1 \rightarrow \tau \\ 
\alpha_1, \alpha_2 \text{ fresh}
\]

What is the type of \text{letfun} f x = f x?
None of the rules discussed so far re-introduces the \( \forall \)-quantor. Is it necessary at all?

```plaintext
let
    f = letfun g x = (x,x) in (f True, f 1)
```

Only necessary if we have `let`-bindings so polymorphic functions can be “exported”
\[
\begin{align*}
T\Gamma \vdash e_1 : \tau & \quad \Gamma \cup \{x : Gen(\Gamma, \tau)\} \vdash e_2 : \tau' \\
\hline
T\Gamma \vdash \text{let}(e_1, x.e_2) : \tau'
\end{align*}
\]

→ The set of all variable occuring free in \( \tau \), but not in \( \Gamma \):

- \( Gen(\Gamma, \tau) = \forall (TV(\tau) \setminus TV(\Gamma)) \)
Specific rules for plus, mult, inl, can be derived from their type is in $\Gamma$ and the application rule.

**Example:**

\[
inl : \forall a. \forall b. a \rightarrow (a + b)
\]

What is the type of $\text{inl}(e)$ for an arbitrary expression $e$?
the inference rules describe Robin Milner’s type inference algorithm $W$
returns the same typing scheme as the non-syntax directed rules discussed previously (modulo $\forall$-quantification)
Simple unification algorithm:

- input: two type terms $t_1$ and $t_2$, forall quantified variables replaced by fresh, unique variables
- output: the most general unifier of $t_1$ and $t_2$ (if it exists)
Cases: $t_1$ and $t_2$

1. both are type variables $v_1$ and $v_2$:
   - if $v_1 = v_2$, return the empty substitution
   - otherwise, return $[v_1/v_2]$

2. both are primitive types
   - if they are the same, return the empty substitution
   - otherwise, there is no unifier

3. both are product types, with $t_1 = (t_{11} \times t_{12})$, $t_1 = (t_{21} \times t_{22})$
   - compute the mgu $S$ of $t_{11}$ and $t_{21}$
   - compute the mgu of $S'$ of $S \ t_{12}$ and $S \ t_{22}$
   - return $S' \cup S$

4. function types / sum types (see product types)

5. only one is a type variable $v$, the other an arbitrary type term $t$
   - if $v$ occurs in $t$, there is no unifier
   - otherwise, return $[t/v]$

6. otherwise, there is no unifier
Implement type inference for MinHs:

- unification
- data type for substitution and operations on this type (see inference rules)
- free variable check
- type inference algorithm
data Bind = Bind Id (Maybe Type) [Id] Exp
    deriving (Read, Show, Eq)

tyInfBnd :: TypeEnv -> Bind -> TC (Bind, Type, Subst)
tyInfExpr :: TypeEnv -> Exp -> TC (Exp, Type, Subst)
Representation of Types:

xo

data Type
  = TyVarTY TyVar
  | FunTy Type Type
  | ForallTy TyVar Type -- a polymorphic type
  | TyApp Type Type -- application
  | TyConstr TyCon -- regular type constructor

data TyCon
  = UnitCon
  | BoolCon
  | IntCon
  | PairCon
  | SumCon
Some Examples:

(we represent terms of type `Id` as string here, in reality, the representation is more complicated but irrelevant for the assignment)

<table>
<thead>
<tr>
<th>Type</th>
<th>Term Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a \to b$</td>
<td><code>FunTy (TyVarTy &quot;a&quot;) (TyVarTy &quot;b&quot;)</code></td>
</tr>
<tr>
<td>$\forall a. a \to a$</td>
<td><code>Forall &quot;a&quot; (FunTy (TyVarTy &quot;a&quot;) (TyVarTy &quot;a&quot;))</code></td>
</tr>
<tr>
<td><code>Int</code></td>
<td><code>TyConstr IntCon</code></td>
</tr>
<tr>
<td><code>(Int, Bool)</code></td>
<td><code>TyApp (TyApp (TyConstr PairCon) (TyConstr IntCon)) (TyConstr BoolCon)</code></td>
</tr>
</tbody>
</table>
THE TC MONAD

→ We need a supply of fresh names
→ Have to keep track of the names already used
→ Haskell does not have global variables or a global state

The function `freshName:: TC Id` returns a new identifier, wrapped in the `TC` type (compare to `Maybe` type)

There is only one way (you should use) to access the “contents” of a `TC` type, the `<-` operator:

→ `new_ident <- freshName`

unwraps the `TC` type returned by `freshName` and binds the content to `new_ident`

Note that the type of the whole expression `new_ident <- freshName` is again `TC Id`
More operations and notation:

- **return::** \( a \rightarrow TC\ a \) to wrap a value of type \( a \) into \( TC\ a \)
- **let x = expr** when you want to bind a value of non-TC type \( a \) to a variable. This is different to usual let-bindings: if \( expr \) has type \( a \), \( let\ x = expr \) has type \( TC\ a \)
- **the do-notation allows to evaluate a sequence of expressions of type \( TC\ a \):**

```haskell
foo :: TC Int -- a useless function
foo =
    do
        new_id <- freshName -- :: TC Id
        let x = 5 * 3 -- :: TC Int
        return x -- :: TC Int
```
Note that, given these operations, once we call `freshName` anywhere in a function, the whole function will have the result type `TC`, as there is no way to get rid of the `TC` constructor.
Example:

We implement the forall elimination rule:

\[
\frac{ x : \forall a_1 \ldots \forall a_n. \tau \in \Gamma }{ \Gamma \vdash x : [\beta_1/a_1] \ldots [\beta_n/a_n] \tau, \quad \beta_i \text{ fresh} }
\]

which takes a type term, removes all forall-quantifiers, and replaces all the bound variables by fresh variables:

If type starts with \texttt{ForallTy}:

1. generate fresh type variable
2. replace all occurrences of bound variable with the new variable using substitution
3. call \texttt{forallElim} recursively on the result in case there are more quantifiers
4. return result

Otherwise, simply return original type: there is nothing
Definition (type annotation of each expression in comments):

forallElim:: Type -> TC Type
forallElim (ForallTy i t) =
  do
    newId <- freshName -- :: TC Id
    let newT = applySubst [(i, TyVarTy newId)] t -- :: TC Type
    let resultT = forallElim newT -- :: TC Type
    return resultT -- :: TC Type
forallElim t = return t

Assuming that a substitution is defined to be list of (Id, Type),
and the function applySubst:: Subst -> Type -> Type
applies a substitution to a type
Binding the result of the recursive call to resultT is not necessary, we could just write:

```haskell
forallElim :: Type -> TC Type
forallElim (ForallTy i t) =
  do
    newId <- freshName -- :: TC Id
    let newT = applySubst [(i, TyVarTy newId)] t -- :: TC Type
    forallElim newT -- :: TC Type

forallElim t = return t
```
**TYPE CLASSES AND OVERLOADING**

- We add the type `Float` to `MinHs`
- How does this affect the type of the built-in arithmetic operations?

**Idea:**

- Group types together which share some properties and operations into a **class** of types
  - `Num` denotes the class of numerical types which work with arithmetic operations
  - `Eq` is the class of types whose elements can be compared using `==`
We write

$\Rightarrow$ $\text{Num } t$ to indicate that a type $t$ is a member of the type class $\text{Num}$, and

$\Rightarrow$ $f :: \text{Num } t \Rightarrow \tau$ to say that $f$ has the type $\tau$ under the condition that $t$ is a member of the type class $\text{Num}$
Predicates \( \pi \ ::= D \tau \)

Polytypes \( \sigma \ ::= \pi \Rightarrow \sigma | \tau | \forall t. \sigma \)

Monotypes \( \tau \ ::= t | \ldots \)

Expressions \( e \ ::= \text{Fun } t \text{ in } e | \text{inst}(e, \tau) | \ldots \)

Values \( v \ ::= \text{Fun } t \text{ in } e | \ldots \)

where \( D \) are class names
New type of operations:

\[ (+) :: \forall a. \text{Num} \ a \Rightarrow a \to a \to a \]

\[ \ldots \]

\[ (==) :: \forall a. \text{Eq} \ a \Rightarrow a \to a \to \text{Bool} \]
New type of operations:

\[\rightarrow (\) \, \forall a. \text{Num} a \Rightarrow a \rightarrow a \rightarrow a \]

\[\rightarrow \ldots\]

\[\rightarrow (==) \, \forall a. \text{Eq} a \Rightarrow a \rightarrow a \rightarrow \text{Bool}\]

Note that

\[1.0 + 1\]

is not possible since addition requires both arguments to be of the same type!
For type inference, we need to know which types are in which class:

- Num Int
- Num Float
- Eq Int
- Eq Float
- Eq Bool
- \( \forall a. \forall b. \text{Eq } a \Rightarrow \text{Eq } b \Rightarrow \text{Eq}(a, b) \)

Let \( P \) be the set of predicates
\[
\{ \text{Num Int, ... } \forall a. \forall b. \text{Eq } a \Rightarrow \text{Eq } b \ldots \}
\]
Inferring Predicates

Given a predicate set $P$, we say $P$ entails a constraint $c$ (written $P \models c$) if and only if

1. $c \in P$, or
2. $P \models \forall a. c'$ and $c = [t/a]c'$, or
3. $P \models \pi \Rightarrow c$ and $P \models \pi$
Type Inference

Previous rules stay as they are, we just add $P$

$$
\frac{x : \tau \in \Gamma}{P \vdash x : \tau}
\quad
\frac{P \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{P \vdash \text{apply}(e_1, e_2) : \tau_2}
\frac{P \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{P \vdash \text{pair}(e_1, e_2) : (\tau_1 \times \tau_2)}
\frac{P \vdash e : \forall t.\tau}{\Gamma \vdash e : [\tau_1/t]\tau}
\frac{P \vdash e : \tau \quad t \notin \text{TV}(\Gamma)}{\Gamma \vdash e : \forall t.\tau}
$$
We need two additional rules:

- **Elimination:**
  
  \[
  \begin{align*}
  & P \mid \Gamma \vdash e : \pi \Rightarrow \rho \quad P \vdash \pi \\
  \quad \Rightarrow & \quad \frac{P \mid \Gamma \vdash e : \rho}{P \mid \Gamma \vdash e : \pi \Rightarrow \rho}
  \end{align*}
  \]

- **Introduction:**
  
  \[
  \begin{align*}
  & P, \pi \mid \Gamma \vdash e : \rho \\
  \quad \Rightarrow & \quad \frac{P \mid \Gamma \vdash e : \pi \Rightarrow \rho}{P \mid \Gamma \vdash e : \pi \Rightarrow \rho}
  \end{align*}
  \]
Let \{ (+) :: \forall a. \text{Num } a \Rightarrow a \to a \to a \} \subseteq \Gamma \text{ and } \{ \text{Num Int} \} \subseteq P.

The \Rightarrow-\text{Elimination rule is, for example, necessary to infer}

\text{plus} :: \text{Float} \to \text{Float} \to \text{Float}:

1. \text{plus} :: \forall a. \text{Num } a \Rightarrow a \to a \to a \text{ (since this type is in } \Gamma, \text{ this implies)}

2. \text{plus} :: \text{Num Float} \Rightarrow \text{Float} \to \text{Float} \to \text{Float} \text{ (\forall-elimination rule), this implies)}

3. \text{plus} :: \text{Float} \to \text{Float} \to \text{Float}, (\Rightarrow-\text{elimination rule, since } P \vdash \text{Num Float})