Simple unification algorithm:

- input: two type terms $t_1$ and $t_2$, for all 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:

```plaintext
xo

data Type
  = TyVar 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 $\text{Id}$ 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>$a \rightarrow b$</td>
<td>$\text{FunTy} (\text{TyVarTy} &quot;a&quot;) (\text{TyVarTy} &quot;b&quot;)$</td>
</tr>
<tr>
<td>$\forall a. \ a \rightarrow a$</td>
<td>$\text{Forall} &quot;a&quot; \ (\text{FunTy} (\text{TyVarTy} &quot;a&quot;) (\text{TyVarTy} &quot;a&quot;))$</td>
</tr>
<tr>
<td>$\text{Int}$</td>
<td>$\text{TyConstr IntCon}$</td>
</tr>
<tr>
<td>$(\text{Int}, \text{Bool})$</td>
<td>$\text{TyApp} (\text{TyApp} (\text{TyConstr PairCon})$</td>
</tr>
<tr>
<td></td>
<td>$\text{(TyConstr IntCon)}) (\text{TyConstr BoolCon})$</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 -> 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:

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

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 occurences 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

THE TC MONAD
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

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

- $f :: \text{Num } t \Rightarrow \tau$ 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 \[ \sigma ::= \pi \Rightarrow \sigma \mid \tau \mid \forall t. \sigma \]

Monotypes \[ \tau ::= t \mid \ldots \]

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

Values \[ v ::= \text{Fun } t \text{ in } e \mid \ldots \]

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

→ (+) :: ∀ a. Num a ⇒ a → a → a
→ ...
→ (==) :: ∀ a. Eq a ⇒ a → a → Bool
New type of operations:

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

\[ \ldots \]

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

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`
- `∀ a. ∀ b. Eq a ⇒ Eq b ⇒ Eq(a, b)`

Let $P$ be the set of predicates

$\{\text{Num Int, ... } ∀ a. ∀ b. \text{Eq } a ⇒ \text{Eq } b \text{ ...}\}$
INFERRING PREDICATES

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

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

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

\[\frac{x : \tau \in \Gamma}{P | \Gamma \vdash x : \tau}\]

\[\frac{P | \Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{P | \Gamma \vdash \text{apply}(e_1, e_2) : \tau_2}\]

\[\frac{P | \Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{P | \Gamma \vdash \text{pair}(e_1, e_2) : (\tau_1 \times \tau_2)}\]

\[\frac{P | \Gamma \vdash e : \forall t.\tau}{\Gamma \vdash e : [\tau_1/t]\tau}\]

\[\frac{P | \Gamma \vdash e : \tau \quad t \notin TV(\Gamma)}{\Gamma \vdash e : \forall t.\tau}\]
We need two additional rules:

⇒ ▹- Elimination:

\[
\frac{P \mid \Gamma \vdash e : \pi \Rightarrow \rho \quad P \vdash \pi}{P \mid \Gamma \vdash e : \rho}
\]

⇒ ▹- Introduction:

\[
\frac{P, \pi \mid \Gamma \vdash e : \rho}{P \mid \Gamma \vdash e : \pi \Rightarrow \rho}
\]
Let \{(+) :: \forall a. \textit{Num} \ a \Rightarrow a \rightarrow a \rightarrow a \} \subseteq \Gamma \text{ and } \{\textit{Num} \ 	extit{Int}\} \subseteq P.

The $\Rightarrow$- Elimination rule is, for example, necessary to infer \texttt{plus} :: \textit{Float} \rightarrow \textit{Float} \rightarrow \textit{Float}:

1. \texttt{plus} :: \forall a. \textit{Num} \ a \Rightarrow a \rightarrow a \rightarrow a \text{ (since this type is in $\Gamma$, this implies)}
2. \texttt{plus} :: \textit{Num} \textit{Float} \Rightarrow \textit{Float} \rightarrow \textit{Float} \rightarrow \textit{Float} \text{ ($\forall$- elimination rule), this implies)
3. \texttt{plus} :: \textit{Float} \rightarrow \textit{Float} \rightarrow \textit{Float}, \text{($\Rightarrow$-elimination rule, since $P \models \textit{Num} \textit{Float}$)}