1 Constraint solving task

We will consider the task of solving a single subtyping constraint given the program's subtype declarations.

I formulate a simple language to specify this task precisely, using somewhat Prolog-like notation:

Program	P ::=	$\overline{D}$	
Declaration	D ::=	$a(\overline{\tau}) \le b(\overline{\sigma})$:- $\overline{C}$.	$a(\overline{\tau})$ subtypes $b(\overline{\sigma})$ if $\overline{C}$ can be proven
Constraint	C ::=	$\tau \le \sigma$	
Type	τ, σ ::=	$T \mid a(\overline{\tau})$	type is either type variable or "named type"
Type Name	a, b, t, ...		names (essentially type constructors) are lower-case
Type Variable	T, S, X, ...		type variables are capitalized

I use the "vector notation": $\overline{D}$ is $D_1, \ldots, D_n$ for some $n \ge 0$; D and $\overline{D}$ are different variables. Note that constraints and declarations may contain type variables; $ftv(D)$ or $ftv(C)$ are functions that return these ("free") type variables.

The task is:

given program $P = \overline{D}_{prog}$ and constraint C with $ftv(C) = \overline{X}$, decide whether $\exists\overline{X}.\ C$

In other words, can C be proven from $\overline{D}_{prog}$, possibly instantiating $\overline{X}$.

A constraint C can be "proven" (or "constraint holds") in one of the following cases:

- there is a declaration $(a(\overline{\tau}) \le b(\overline{\sigma})$:- $\overline{C}.) \in \overline{D}_{prog}$ whose type variables can be instantiated so that:
  - $a(\overline{\tau}) \le b(\overline{\sigma})$ instantiates into C, and
  - the instantiated $\overline{C}$ can in turn be proven
- C is obtained by transitivity of subtyping from other constraints that can be proven (i.e., if $\tau_1 \le \tau_2, \tau_2 \le \tau_3, \ldots, \tau_{n-1} \le \tau_n$ then $\tau_1 \le \tau_n$)
- C holds by reflexivity (i.e., each type subtypes itself)

2 Examples of program declarations

I use Vortex as my data-mining body of subtype declarations. (I'd be happy if I can prove algorithm termination for these declarations or at least a majority of them.) While that's a lot of code, the subtype declarations follow very few patterns. Here is my collection of typical declarations.

Most are unconditional:

```
int ≤ num.
string ≤ indexed(char).
method_table ≤ hash_table(msg_key,method_set).
list(T) ≤ sequence(T).
table(K,V) ≤ collection(V).
indexed(T) ≤ table(int,T).
```

Lots of collections are covariant:

```
list(T) ≤ list(S) :- T ≤ S.
table(K,V1) ≤ table(K,V2) :- V1 ≤ V2.
```

A few types have "associated constraints," e.g., you are not allowed to say `set(T)` unless `T` is comparable. These associated constraints are always listed whenever the declaration mentions such types. I will list them after ';' at the end of the list:

```
set(T) ≤ unordered_collection(T) :- ; T ≤ comparable(T).
keyed_set(K,V) ≤ table(K,V) :- ; K ≤ comparable(K), V ≤ keyed_comparable(K).
set(T) ≤ set(S) :- T ≤ S; T ≤ comparable(T), S ≤ comparable(S).
```

F-bounded types (I call them "comparables") are contravariant and form "hierarchies":

```
comparable(T) ≤ comparable(S) :- S ≤ T; T ≤ comparable(T), S ≤ comparable(S).
ordered(T) ≤ comparable(T) :- ; T ≤ ordered(T).
```

Lots of declarations indicate "instantiations" of comparables. This is where a vast majority of truly conditional subtyping occurs:

```
num ≤ ordered(T).
sequence(T) ≤ comparable(sequence(T)) :- T ≤ comparable(T).
table(K,V) ≤ comparable(table(K,V)) :- V ≤ comparable(V).
keyed_set(K,V) ≤ comparable(keyed_set(K,V) :-
   V ≤ comparable(V); K ≤ comparable(K), V ≤ keyed_comparable(K).
```

Finally there are monsters from the IPCA framework where types' associated constraints introduce type variables not mentioned in the types. But other than that, they follow the same patterns:

```
ipca_function_node(C) ≤ ipca_call_graph_node(C) :-
   ; C ≤ ipca_algorithm_node(C,S), S ≤ class_set(S,D), D ≤ hashable(D).
ipca_call_graph_node(C) ≤ call_graph_node(ipca_call_graph_node(C)) :-
   ; C ≤ ipca_algorithm_node(C,S), S ≤ class_set(S,D), D ≤ hashable(D).
```

3 The solving algorithm

I'd like to give only an informal intuition for how my algorithm works. Firstly, it keeps a worklist of constraints that need to be proven (seeded by the initial constraint), along with the type variables which are free in these constraints. These type variables are considered existentially quantified and are subject to substitution as needed.

Secondly, the algorithm removes from the worklist any constraint (the leftmost constraint, in practice) which has named types (i.e., not type variables) on both left and right side, for example, list(pair(X,Y) £ comparable(list(pair(X,Y)). Then it builds all subtyping chains between `list` and `comparable` by transitivity from the program's declarations. For example, the following of above declarations would be chained together:

```
list(T) ≤ list(S) :- T ≤ S.
list(T) ≤ sequence(T).
sequence(T) ≤ comparable(sequence(T)) :- T ≤ comparable(T).
comparable(T) ≤ comparable(S) :- S ≤ T; T ≤ comparable(T), S ≤ comparable(S).
```

forming the following "combined" declaration:

```
list(A) ≤ comparable(B) :-
   A ≤ C, C ≤ comparable(C), B ≤ sequence(C),
   ; sequence(C) ≤ comparable(sequence(C)), B ≤ comparable(B).
```

Observe that free type variables of each declaration (which are equivalent up to alpha-renaming) are replaced with fresh, existentially quantified type variables, then unification occurs to match the adjacent types from adjacent declarations. For example, `T` from the third declaration in the chain is replaced with fresh `C` and then `sequence(C)` is unified with the `T` from the fourth declaration.

Thirdly, this combined declaration is used to prove our constraint just the way Prolog does that. Namely, the declaration is unified with the constraint, in our case resulting in A := pair(X,Y), B := list(pair(X,Y)); C is added to the algorithm's list of free type variables, and the instantiated sub-constraints added to the algorithm's constraint list: pair(X,Y)≤C, C≤comparable(C), list(pair(X,Y))≤sequence(C), sequence(C) ≤comparable(sequence(C)), list(pair(X,Y))≤comparable(list(pair(X,Y)). If this combined declaration does not lead to success, others (if any) are tried via backtracking.

As a side note, observe that in order to prove list(pair(X,Y))≤comparable(list(pair(X,Y)) we must again prove it. This means that we need to take a fixpoint of our subtype relation. However, this can be done once (manually, in a sense) for declarations like comparable(T)≤comparable(S) :- S≤T; T≤comparable(T), S≤comparable(S) by removing the last sub-constraint.

By following the second and third steps repeatedly we will eventually reach the situation where each of the remaining constraints on the worklist has at least one type variable in it. The algorithm picks one of such variables (quite arbitrarily) and the constraints which bound it. One of the two situations then occurs in practice:

- the variable has some lower bounds $\bar{\tau}$ and some upper bounds $\bar{\sigma}$ none of which mention the variable. Then the variable is eliminated and the transitive closure constraints $\{\tau \leq \sigma \mid \tau \in \bar{\tau}, \sigma \in \bar{\sigma}\}$ are added to the worklist.
- the bounds look like $\tau \leq X$, $X \leq$ comparable(X). Then X is substituted with the lower, non-recursive bound, τ, and the transitive closure constraint $\tau \leq$ comparable(τ) is added to the worklist. (I haven't seen cases in Vortex where this would miss a solution!)