Last time, we saw that the unification operation \text{Unify}(\tau_1, \tau_2)$ is used in type checking to “make two types the same”. The two types may have unification variables $?_i$ and $?_j$. For example, if we know that a variable \textit{x} has type $?_1 \rightarrow \text{int}$ and we later learn that it has type \textit{string} $\rightarrow$ $?_2$, we will unify these two types and learn that $?_1 = \text{string}$ and $?_2 = \text{int}$. We express this as a substitution, which we’ll represent with an association list (like we used to represent environments in HW3), so the substitution resulting from this unification would be \([(?_1, \text{string}); (?_2, \text{int})]\).

The basic idea of unification is to proceed recursively through the two types and try to match like types with like types. Types are basically ASTs, so this is like a recursion down two trees at the same time, and we’re trying to match up all of the leaves (which are base types like \text{int} and unification variables). If we try to unify a unification variable with a type, we wind up with a substitution for that unification variable.

There are two ways unification can fail. The most obvious is if we try to unify two types that obviously can’t be unified because they just “look different”. An example would be \text{Unify}(\text{int}, \text{string}). There’s no way to make these two the same. Another examples is \text{Unify}(?_1 \rightarrow ?_2, ?_3 * ?_4). Even though there are a lot of unification variables, there’s no way to make these two types the same because one is an arrow and one is a product: they have different “shapes”. Because of this, this type of failure is sometimes called a shape mismatch.

The other type of unification failure is more subtle. Let’s say we’re trying to type check the function \text{let rec f x = f f}. We know \textit{f} is a function, so we’ll give it the type $?_1 \rightarrow ?_2$. Since we pass \textit{f} to \textit{f}, that means that \textit{f} needs to have the right type for the argument, which is $?_1$. We already know that \textit{f} has type $?_1 \rightarrow ?_2$, so this means we’ll call \text{Unify}(?_1, ?_1 \rightarrow ?_2). But there’s no possible substitution that can make these two equal, for the same reason that there’s no value of \textit{x} that can make the equation \textit{x} = \textit{x} + 1 true\footnote{Don’t confuse this with the statement \textit{x} = \textit{x} + 1 as you might write in an imperative language; we’re not setting anything here, we’re asserting that two things are equal that can’t possibly be equal.}. The problem is that $?_1$ occurs inside $?_1 \rightarrow ?_2$. So whenever we unify a unification variable $?_i$ with some type \tau (that isn’t just the unification variable itself; \text{Unify}(?_i, ?_i) of course has no problem at all), we need to make sure that $?_i$ doesn’t occur in (appear anywhere inside) \tau. This is called the occurs check. If the occurs check fails (i.e., $?_i$ appears in \tau), then unification is not possible.

Here’s the full algorithm:

\textbf{Algorithm:} \text{Unify}(\tau_1, \tau_2)

\textbf{Returns a substitution or an error.}

1. If \tau_1 and \tau_2 are both the same base type (\text{int}, \text{string}, \text{bool} or \text{unit}), return the empty list [].
2. If \tau_1 = \tau_2 = $?_i$ (i.e., they are the same unification variable), return [].
3. If \tau_1 = $?_i$, then:
   (a) If $?_i$ occurs in \tau_2, then error.
   (b) Otherwise, return \([(?_i, \tau_2)]\).
4. If \tau_2 = $?_i$, then:
(a) If ?₁ occurs in τ₁, then error.
(b) Otherwise, return [(?₁, τ₁)].

5. If τ₁ = τ′₁ list and τ₂ = τ′₂ list, then return Unify(τ′₁, τ′₂)

6. If τ₁ = τ′₁ → τ′′₁ and τ₂ = τ′₂ → τ′′₂ then let σ = Unify(τ′₁, τ′₂) and return σ concatenated with Unify([(σ|τ′₁), (τ′₂|σ)]).

7. Similar to above for if τ₁ = τ′₁ * τ′′₁ and τ₂ = τ′₂ * τ′′₂

8. Otherwise, error.

Case 6 above is worth discussing in more detail. When we unify τ′₁ and τ′₂, we get a substitution σ, which gives us several substitutions we need to perform. We need to perform those substitutions immediately on τ′′₁ and τ′′₂ before unifying them, because σ might tell us something like “replace ?₀ with unit”, and τ′₁ might have ?₀ in it. To make this clearer, suppose τ₁ = int → string and τ₂ =?= → ?₀. Then, we have τ₁ = int, τ₂ =?= =?=, τ′₁ = string and τ′₂ =?= =?=0. When we unify int and ?₀, we get the substitution [(?₀, int)]. If we then go ahead and unify string and ?₀, we’d get the substitution [(?₀, string)]. Now, we have to somehow reconcile these two substitutions, which of course isn’t possible because ?₀ can’t be both int and string. Instead, we unify [(?₀, int)][string = string and [(?₀, int)]][?₀ = int, which gives us an error (which is what we want because these two types can’t be unified.)

The file unify.ml contains some definitions that are used in unification. As we said above, substitutions are just association lists:

definition substitution = (int * typ) list

In that definition, ?₀ is represented as just the integer 0. We also include a function sub_all : substitution -> typ -> typ, where sub_all s t computes [s|t].

For each of the following pairs of types, say whether or not the two types can be unified. If they can, give the substitution that results. If not, briefly (in one sentence or so) describe why not.

(a) int * ?₁   ?₂
(b) int * ?₁   ?₂ * string
(c) int * ?₁   ?₁ * string
(d) int -> ?₁  string -> ?₂
(e) ?₁ -> ?₂   ?₃ * ?₄
(f) ?₁ list   ?₂ list list
(g) ?₁ list   ?₁

Answers:

(a) (?₂, int * ?₁)
(b) (?₁, string), (?₂, int)
(c) No. ?₁ would have to unify with both int and string.
(d) No. int doesn’t unify with string.
(e) No. Arrow doesn’t unify with product.
(f) (?₁, ?₂ list)
(g) No. ?₁ occurs in ?₁ list.