IIT CS440: Programming Languages and Translators

Lecture 15: Unification

Prof. Stefan Muller

Monday, Mar. 6, 2023

Last time, we saw that the *unification* operation Unify(τ_1, τ_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 x has type $?_1$ `->` `int` and we later learn that it has type `string` `->` $?_2$, we will unify these two types and learn that $?_1$ = `string` and $?_2$ = `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 $[(?_1, \texttt{string}); (?_2, \texttt{int})]$.

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 `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 Unify(`int`, `string`). There's no way to make these two the same. Another examples is Unify($?_1$ `->` $?_2$, $?_3$ `*` $?_4$). 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 `let rec f x = f f`. We know `f` is a function, so we'll give it the type $?_1$ `->` $?_2$. Since we pass `f` to `f`, that means that `f` needs to have the right type for the argument, which is $?_1$. We already know that `f` has type $?_1$ `->` $?_2$, so this means we'll call Unify($?_1$, $?_1$ `->` $?_2$). But there's no possible substitution that can make these two equal, for the same reason that there's no value of x that can make the equation $x = x + 1$ true[1] The problem is that $?_1$ *occurs inside* $?_1$ `->` $?_2$. So whenever we unify a unification variable $?_i$ with some type τ (that isn't just the unification variable itself; Unify($?_i$, $?_i$) of course has no problem at all), we need to make sure that $?_i$ doesn't occur in (appear anywhere inside) τ. This is called the *occurs check*. If the occurs check fails (i.e., $?_i$ appears in τ), then unification is not possible.

Here's the full algorithm:

Algorithm: Unify(τ_1, τ_2)
Returns a substitution or an error.

1. If τ_1 and τ_2 are both the same base type (`int`, `string`, `bool` or `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 τ_2, then error.
   (b) Otherwise, return $[(?_i, \tau_2)]$.
4. If $\tau_2 = ?_i$, then:

[1] Don't confuse this with the statement `x = 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.

(a) If $?_i$ occurs in τ_1, then error.

(b) Otherwise, return $[(?_i, \tau_1)]$.

5. If $\tau_1 = \tau_1'$ `list` and $\tau_2 = \tau_2'$ `list`, then return Unify(τ_1', τ_2')

6. If $\tau_1 = \tau_1'$ `->` τ_1'' and $\tau_2 = \tau_2'$ `->` τ_2'' then let σ = Unify(τ_1', τ_2') and return σ concatenated with Unify($[\sigma]\tau_1''$, $[\sigma]\tau_2''$).

7. Similar to above for if $\tau_1 = \tau_1'$ `*` τ_1'' and $\tau_2 = \tau_2'$ `*` 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 τ_1 = `int -> string` and $\tau_2 = ?_0$ `->` $?_0$. Then, we have τ_1' = `int`, $\tau_2'' = ?_0$, τ_1'' = `string` and $\tau_2'' = ?_0$. When we unify `int` and $?_0$, we get the substitution $[(?_0, \texttt{int})]$. If we then go ahead and unify `string` and $?_0$, we'd get the substitution $[(?_0, \texttt{string})]$. 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 $[[(?_0, \texttt{int})]]\texttt{string} = \texttt{string}$ and $[[(?_0, \texttt{int})]]?_0 = \texttt{int}$, 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:

```
type 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) $(?_2, \texttt{int} * ?_1)$

(b) $(?_1, \texttt{string}), (?_2, \texttt{int})$

(c) No. $?_1$ would have to unify with both int and string.

(d) No. int doesn't unify with string.

(e) No. Arrow can't unify with product.

(f) $(?_1, ?_2\ \texttt{list})$

(g) No. $?_1$ occurs in $?_1$ `list`.