A Unification Algorithm (for type expressions)

Input: Two terms $T_1$ and $T_2$ to be unified
Output: $\emptyset$, the most general unifier of $T_1$ and $T_2$, or failure.

Algorithm:

Initialize the substitution $\emptyset$ to be empty, the stack to contain the equation $T_1 = T_2$, and failure to false.

While stack not empty and no failure do

pop $X = Y$ from the stack

case

$X$ is a variable that does not occur in $Y$:

substitute $Y$ for $X$ in the stack and in $\emptyset$.
add $X = Y$ to $\emptyset$.

$Y$ is a variable that does not occur in $X$:

substitute $X$ for $Y$ in the stack and in $\emptyset$.
add $Y = X$ to $\emptyset$.

$X$ and $Y$ are identical constants or variables:

continue

$X$ is $X_1 \rightarrow X_2$ and $Y$ is $Y_1 \rightarrow Y_2$ for some terms $X_1$, $X_2$, $Y_1$, $Y_2$,
OR $X$ is $X_1 \text{ list}$ and $Y$ is $Y_1 \text{ list}$ for some terms $X_1$, $Y_1$:

push $X_i = Y_i$ on the stack, for each appropriate $X_i$ and $Y_i$.
otherwise:

failure := true;

end while

if failure, then output failure
else output $\emptyset$
• A substitution is a finite set (possibly empty) of pairs of the form \( X_i = t_i \) where \( X_i \) is a variable and \( t_i \) is a term, and \( X_i \neq X_j \) for every \( i \neq j \) and \( X_i \) does not occur in \( t_j \) for any \( i \) and \( j \).

  - The result of applying a substitution \( \theta \) to a term \( A \), denoted by \( A\theta \), is the term obtained by replacing every occurrence of \( X \) in \( A \) by \( \theta \), for each pair \( X = \tau \) in \( \theta \).

• A term \( A \) is an instance of a term \( B \) if there is a substitution \( \theta \) such that \( A = B\theta \).

  - \texttt{int→int list} is an instance of \( T_1\rightarrow T_2 \), in which \( \theta \) = \{ \texttt{T_1 = int, T2 = int list} \}

• A term \( A \) is a common instance of terms \( B \) and \( C \) if there are substitutions \( \theta_1 \) and \( \theta_2 \) such that \( A = B\theta_1 \) and \( A = C\theta_2 \).

6.1.1 Unification

• A term \( s \) is more general than a term \( t \) if \( t \) is an instance of \( s \) but \( s \) is not an instance of \( t \). A term \( s \) is an alphabetic variant of a term \( t \) if both \( s \) is an instance of \( t \) and \( t \) is an instance of \( s \). Alphabetic variants can be converted to one another simply by renaming variables.

• A unifier of two terms \( t_1 \) and \( t_2 \) is a substitution \( \theta \) making the terms identical, \( t_1\theta = t_2\theta \). If a unifier of two terms exists, than the terms are said to unify.

• A most general unifier (or mgu) of two terms \( t_1 \) and \( t_2 \) is a substitution \( \theta \) that unifies \( t_1 \) and \( t_2 \) such that the common instance \( t_1\theta \) is more general than any other common instance of \( t_1 \) and \( t_2 \). That is, for any other unifier \( \theta' \) of \( t_1 \) and \( t_2 \), \( t_1\theta \) is more general than \( t_1\theta' \). If two terms unify, there is a unique most general unifier (up to the renaming of variables).

• A unification algorithm computes the most general unifier for two terms. An algorithm for unification is as follows:
5.1. Application

\[
\frac{A \vdash c_1 : T_1 \rightarrow T_2, \ A \vdash c_2 : T_1}{A \vdash e_1 \ e_2 : T_2}
\]

5.2. Lambda Abstraction (Function Abstraction)

"Assumptions A plus x has type T_1"

\[
\frac{A \ {\text{x : T}_1} \vdash c : T_2}{A \vdash \text{fn x => c} : T_1 \rightarrow T_2}
\]

ML's "lambda"

5.3. Conditional

\[
\frac{A \vdash c_1 : \text{bool}, \ A \vdash c_2 : T, \ A \vdash c_3 : T}{A \vdash \text{if} \ c_1 \ \text{then} \ c_2 \ \text{else} \ c_3 : T}
\]

5.4. Let

"Assumptions A plus each x_{ij} has type T_{ij}"

\[
\frac{A \vdash c_1 : T_{1_1}, \ldots, \ A \vdash c_n : T_{n_p}, \ A \ {\text{x_{1_{1j}} : T_{1_{1j}}}} \vdash c : T}{A \vdash \text{let val} \ x_1 \ = \ c_{1}, \ \text{val} \ x_2 \ = \ c_2, \ \ldots, \ \text{val} \ x_n \ = \ c_n \ \text{in} \ c \ \text{end} : T}
\]

where x_{ij} is the j^{th} occurrence of x_i in c and T_{1_{ij}} is an instance of T_i.

6. Solving Type Equations using Unification

- During type inference, sets of equations of the form

\[
E_1 = E_2
\]

must be solved. If there is no solution, then there is a type error.

- The left and right hand sides of each equation is constructed out of terms containing constants (such as int and string), type variables (such as T_1 and T_2) and the type constructors \(\rightarrow\) and list.

- The method used to solve the equations is called unification, and was developed in the 1960's by Robinson to facilitate automatic theorem proving. It is the basic computational engine for Prolog.

6.1. Instances and Substitutions

- A term is an expression defined by:

\[
\text{term} = \text{constant | variable | (term \rightarrow \text{term}) | term list}
\]

- A term containing no variables is said to be ground.
Problem?

- $\alpha$ is instantiated two different ways!!

This is OK - For each occurrence of a \texttt{let}-defined variable (such as \texttt{length}), the type variables (such as $\alpha$) in its type can be instantiated differently. This is what allows \texttt{length} to be polymorphic.

- $\alpha$ is called a \texttt{schematic} type variable (also known as a \texttt{generic} type variable).

- When performing type inference, for each occurrence of \texttt{length} in the body of the \texttt{let} expression, replace $\alpha$ with a different variable ($\alpha_1$, $\alpha_2$, etc).

Therefore, the above tree would be:

\[
\begin{array}{c}
+ \quad \text{int} \times \text{int} \rightarrow \text{int} \\
\circ \quad \text{T}_2 \\
\circ \quad \text{T}_3 \\
\text{length: } \alpha_1 \text{ list } \rightarrow \text{int} \\
\text{[1, 2]: int list} \\
\text{length: } \alpha_2 \text{ list } \rightarrow \text{int} \\
\text{["hello"]: string list}
\end{array}
\]

The equations would be:

\[
\alpha_1 \text{ list } \rightarrow \text{int} = \text{int list } \rightarrow \text{T}_2 \\
\alpha_2 \text{ list } \rightarrow \text{int} = \text{string list } \rightarrow \text{T}_3
\]

Solving for the variables would give us:

\[
\begin{align*}
\alpha_1 &= \text{int} \\
\text{T}_2 &= \text{int} \\
\alpha_2 &= \text{string} \\
\text{T}_3 &= \text{int}
\end{align*}
\]

5. Summary of Type Rules

Notation:

\[
\text{A } \vdash \text{x : T}
\]

means "Under type assumptions \text{A} (associating variables with their types), the expression \text{x} has type \text{T}'.'

\[
\frac{R_1, ..., R_n}{R}
\]

means "If $R_1, ..., R_n$ are true, the $R$ is true."

G22.3110 Programming Languages (Honors)
4. The LET construct

RULE #2. Variables defined in a let construct can be used polymorphically in the body of the let.

```ml
let fun length [] = 0
    | length (x::xs) = 1 + length xs
in length [1,2] + length ["hello"]
end;
```

int list → int  string list → int

RULE #3. Occurrences of let-bound variables in the right hand sides of their definitions must be used monomorphically.

```ml
let fun f [x] = 1
    | f (x::xs) = if x = 0 then f [true] + f xs
in ...
end;
```

bool list → int  int list → int  Type Error!

4.1. Example: Type inference of a Let expression

Consider the length program at the top of this page. Using the type inference technique discussed previously, we can determine that length has type: α list → int.

Now we need to infer the type of the body of the let expression, namely

```
length [1,2] + length ["hello"]
```

The tree representation of this expression is:

```
+ @ T2 @ T3
```

```
length: α list → int  [1,2]: int list  length: α list → int  ["hello"]: string list
```

We arrive at the following equations:

\[
\alpha \text{ list } \rightarrow \text{ int } = \text{ int list } \rightarrow T_2
\]

\[
\alpha \text{ list } \rightarrow \text{ int } = \text{ string list } \rightarrow T_3
\]

Using unification, we solve the two equations to get:

\[
\alpha = \text{ int}
\]

\[
T_2 = \text{ int}
\]

\[
\alpha = \text{ string}
\]

\[
T_3 = \text{ int}
\]
Notice that z has two type variables associated with it, T₁ and T₃. Can z have two different types?

**RULE #1.** Use of formal parameters in a function body must be monomorphic: All occurrences of a formal parameter must have the same type.

Therefore, T₁ and T₃ must be the same type. Replacing T₃ by T₁ and replacing T₄ by (T₅ → T₆) in the above equations gives us:

\[
T₀ = T₁ \rightarrow (T₅ \rightarrow T₆) \\
T₂ = T₁ \rightarrow T₅
\]

So x has type T₁ → (T₅ → T₆), y has type (T₁ → T₅), z has type T₁, and the type of the result of f is T₆.

Thus, f has type (T₁ → (T₅ → T₆)) → (T₁ → T₅) → T₁ → T₆.

The restriction (monomorphic use of formal parameters) arises from theoretical problems in type inference, rather than intuition. For example,

```plaintext
let fun f g = (g 1, g true) 
      fun Id x = x 
      in f Id
```

would seem to work fine, returning (1, true). However, this program has a type error because g is used polymorphically (different instances having different types), namely as int → int in the first instance and bool → bool in the second instance.

3. A Mistyping

Consider

```plaintext
fun f a b c = c a (a b c)
```

Using the parse tree representation and trying to solve the equations, we see that no solution can be found:

```
  fun f a : T₀ b : T₁ c : T₂ =

  @  T₆
   @  T₃
   @  T₅
   a : T₀
   b : T₁
   c : T₂
   e : T₂
   a : T₀
   b : T₁
```

\[
T₂ = T₁ \rightarrow T₃ \\
T₀ = T₁ \rightarrow T₂ \\
T₄ = T₂ \rightarrow T₅ \\
T₃ = T₅ \rightarrow T₆
\]

Circular definition: No finite solution!
G22.3110 Programming Languages (Honors)

Polymorphic Type Inference

1. Type Checking and Inference

Static type checking has two goals:

1. To determine if the program is well typed (i.e. catching type errors).
2. To determine the type of each expression. This is necessary for implementation.

Static type inference determines the type of each expression in the absence of declarations.

- For example, one can figure out that the type of map is: \((\alpha \rightarrow \beta) \rightarrow \alpha \text{list} \rightarrow \beta \text{list}\).

\[
\text{fun} \quad \text{map} : \; \alpha \rightarrow \beta \rightarrow \alpha \text{list} \rightarrow \beta \text{list} \\
\text{map} \; f \; (\text{x} : \text{xs}) = f \; \text{x} :: \text{map} \; f \; \text{xs}
\]

2. Finding Types

- Associate type variables with each expression in the program, and solve equations involving those type variables.

For example, given

\[
\text{fun} \; f \; x \; y \; z = x \; z \; (y \; z)
\]

we represent the body of the function in its tree representation, and associate type variables \(T_1 \ldots T_6\) with the nodes in the tree (the symbol @ denotes application):

\[
\text{fun} \; f \; x \; y \; z = \quad \begin{array}{c}
\@ \quad T_6 \\
\@ \quad T_4 & \@ \quad T_3 \\
\quad x \; T_0 & \quad z : T_1 \quad y : T_2 \quad z : T_3
\end{array}
\]

If the application of a function of type \(T_i\) to an argument of type \(T_j\) returns a value of type \(T_k\), then the following equation holds: \(T_i = T_j \rightarrow T_k\). This looks like \(\@ \quad T_k\) in the parse tree representation.

The equations derived from the parse tree above are:

\[
T_0 = T_1 \rightarrow T_4 \\
T_2 = T_3 \rightarrow T_5 \\
T_4 = T_5 \rightarrow T_6
\]

G22.3110 Programming Languages (Honors)