

A Unification Algorithm (for type expressions)

Input: Two terms T_1 and T_2 to be unified

Output: θ , the most general unifier of T_1 and T_2 , or *failure*.

Algorithm:

Initialize the substitution θ 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 θ .

add $X = Y$ to θ .

Y is a variable that does not occur in X :

substitute X for Y in the stack and in θ .

add $Y = X$ to θ .

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_i list and Y is Y_i list for some terms X_i, Y_i :

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 0

- 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 θ to a term A , denoted by $A\theta$, is the term obtained by replacing every occurrence of X in A by t , for each pair $X = t$ in θ .
- A term A is an instance of a term B if there is a substitution θ such that $A = B\theta$.
 - `int \rightarrow int list` is an instance of `T1 \rightarrow T2`, in which $\theta = \{T_1 = \text{int}, T_2 = \text{int list}\}$
- A term A is a common instance of terms B and C if there are substitutions θ_1 and θ_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 θ making the terms identical, $t_1\theta = t_2\theta$. If a unifier of two terms exists, then the terms are said to unify.
- A most general unifier (or mgu) of two terms t_1 and t_2 is a substitution θ 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 θ' 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

$$\frac{A \vdash e_1 : T_1 \rightarrow T_2, \quad A \vdash e_2 : T_1}{A \vdash e_1 e_2 : T_2}$$

5.2. Lambda Abstraction (Function Abstraction)

$$\frac{A [x : T_1] \vdash e : T_2}{A \vdash \mathbf{fn} \ x \Rightarrow e : T_1 \rightarrow T_2}$$

5.3. Conditional

$$\frac{A \vdash e_1 : \mathbf{bool}, \quad A \vdash e_2 : T, \quad A \vdash e_3 : T}{A \vdash \mathbf{if} \ e_1 \ \mathbf{then} \ e_2 \ \mathbf{else} \ e_3 : T}$$

5.4. Let

$$\frac{A \vdash e_1 : T_1, \dots, A \vdash e_n : T_n, \quad A [x_{i,j} : T_{i,j}] \vdash e : T}{A \vdash \mathbf{let} \ \mathbf{val} \ x_1 = e_1 \ \mathbf{val} \ x_2 = e_2 \ \dots \ \mathbf{val} \ x_n = e_n \ \mathbf{in} \ e \ \mathbf{end} : T}$$

where $x_{i,j}$ is the j^{th} occurrence of x_i in e and $T_{i,j}$ 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:

$$\text{term} = \text{constant} \mid \text{variable} \mid (\text{term} \rightarrow \text{term}) \mid \text{term} \ \mathbf{list}$$

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

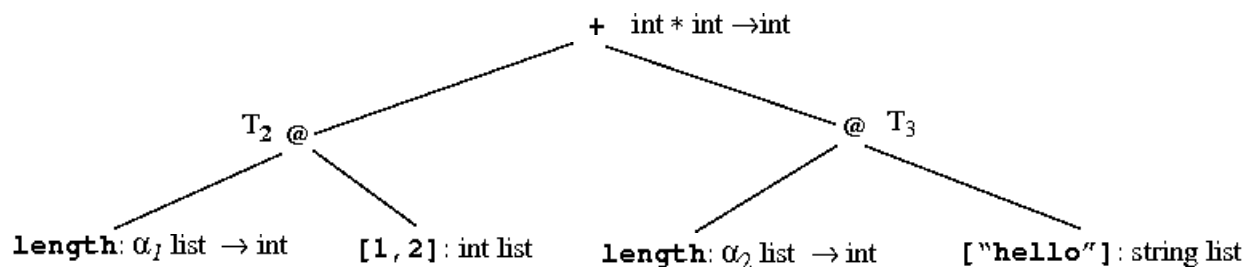
Problem?

- α is instantiated two different ways!!

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

- α is called a **schematic** type variable (also known as a **generic** type variable).
- When performing type inference, for each occurrence of **length** in the body of the let expression, replace α with a different variable (α_1, α_2 , etc).

Therefore, the above tree would be:



The equations would be:

$$\alpha_1 \text{ list} \rightarrow \text{int} = \text{int list} \rightarrow T_2$$

$$\alpha_2 \text{ list} \rightarrow \text{int} = \text{string list} \rightarrow T_3$$

Solving for the variables would give us:

$$\alpha_1 = \text{int}$$

$$T_2 = \text{int}$$

$$\alpha_2 = \text{string}$$

$$T_3 = \text{int}$$

5. Summary of Type Rules

Notation:

$$A \vdash x : T$$

means “Under type assumptions A (associating variables with their types), the expression x has type T”.

$$\frac{R_1, \dots, R_n}{R}$$

means “If R_1, \dots, R_n are true, the R is true.”

4. The LET construct

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

```

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

```

\int $int\ list \rightarrow int$ \int $string\ list \rightarrow int$

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

```

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

```

\int $bool\ list \rightarrow int$ \int $int\ list \rightarrow int$
Type Error!

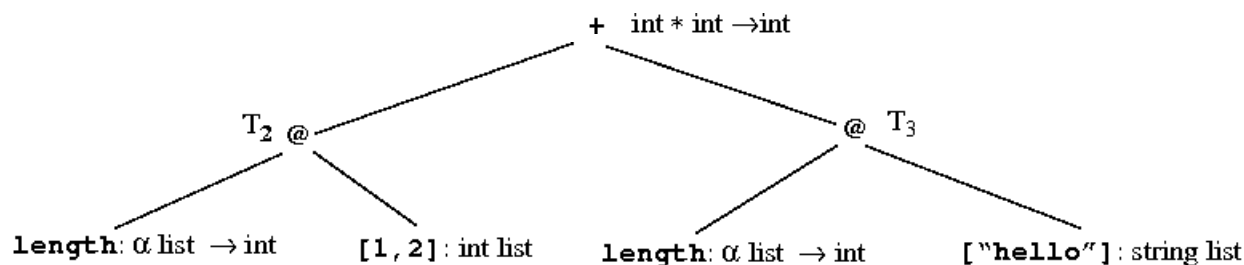
4.1. Example: Type inference of a Let expression

Consider the `length` program at the top of this page. Using the type inference technique discussed previously, we can determine that `length` has type: $\alpha\ list \rightarrow int$.

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:



We arrive at the following equations:

$$\alpha\ list \rightarrow int = int\ list \rightarrow T_2$$

$$\alpha\ list \rightarrow int = string\ list \rightarrow T_3$$

Using unification, we solve the two equations to get:

$$\alpha = int$$

$$T_2 = int$$

$$\alpha = string$$

$$T_3 = int$$

Notice that z has two type variables associated with it, T_1 and T_3 . 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, T_1 and T_3 must be the same type. Replacing T_3 by T_1 and replacing T_4 by $(T_5 \rightarrow T_6)$ in the above equations gives us:

$$\begin{aligned} T_0 &= T_1 \rightarrow (T_5 \rightarrow T_6) \\ T_2 &= T_1 \rightarrow T_5 \end{aligned}$$

So x has type $T_1 \rightarrow (T_5 \rightarrow T_6)$, y has type $(T_1 \rightarrow T_5)$, z has type T_1 , and the type of the result of f is T_6 . Thus, f has type $(T_1 \rightarrow (T_5 \rightarrow T_6)) \rightarrow (T_1 \rightarrow T_5) \rightarrow T_1 \rightarrow T_6$.

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

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

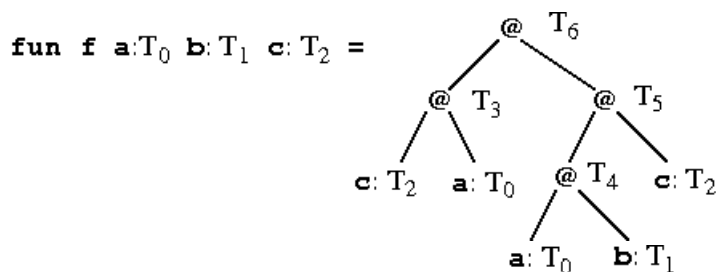
would seem to work fine, returning $(1, \text{true})$. However, this program has a type error because g is used *polymorphically* (different instances having different types), namely as $int \rightarrow int$ in the first instance and $bool \rightarrow bool$ in the second instance.

3. A Mistyping

Consider

```
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:



$$\begin{aligned} T_2 &= T_1 \rightarrow T_3 \\ T_0 &= T_1 \rightarrow T_4 \\ T_4 &= T_2 \rightarrow T_5 \\ T_3 &= T_5 \rightarrow T_6 \end{aligned}$$

$$\longrightarrow T_2 = (T_1 \rightarrow (T_2 \rightarrow T_5)) \rightarrow T_5 \rightarrow T_6$$

Circular definition: No finite solution!

G22.3110 Programming Languages (Honors)

Polymorphic Type Inference

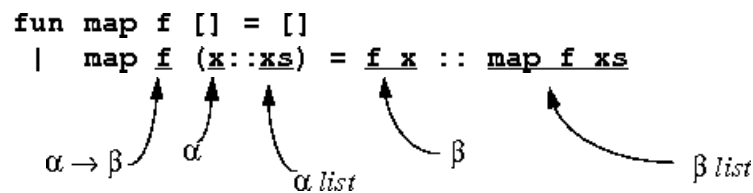
1. Type Checking and Inference

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: $(\alpha \rightarrow \beta) \rightarrow \alpha \text{ list} \rightarrow \beta \text{ list}$.



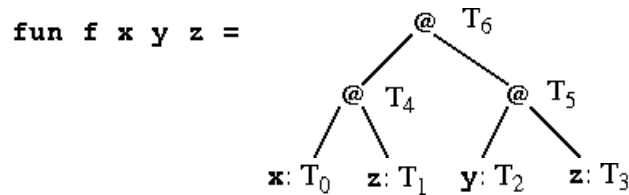
2. Finding Types

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

For example, given

```
fun f x y z = x z (y z)
```

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



If the application of a function of type T_1 to an argument of type T_j returns a value of type T_k , then the

following equation holds: $T_1 = T_j \rightarrow T_k$. This looks like in the parse tree representation.

The equations derived from the parse tree above are:

$$T_0 = T_1 \rightarrow T_4$$

$$T_2 = T_3 \rightarrow T_5$$

$$T_4 = T_5 \rightarrow T_6$$