

Principles of Programming Languages

Type Inference

After having described the process of **type checking**, we now address a more ambitious task in type analysis: **type inference**. In this task, we not only verify that a program is safe, we also allow the programmer to leave some (or even all) the type annotations empty - and attempt to guess their value from the structure of the program.

If we can infer such a consistent set of type annotations for the program, we conclude that the program is type safe and provide the annotations. Otherwise, we point to the inconsistency.

To enable this process, we consider all the type annotations in the *L5* language as **optional**. This process is quite similar to the strategy adopted in TypeScript which we reviewed in Chapter 1 (*gradual typing*).

We first investigate how we can perform the process of type inference manually, and then present an algorithm which automates this process, relying on a critical operation on substitutions called **unification**.

The operation of unification is a fundamental tool in the semantics of programming languages, and we will further expand its use in Chapter 5 (on Logic Programming).

The algorithm we present extends the type checker method by introducing type equations which are constructed for every sub-expression of the expression to be typed.

A solution to the equations assigns types to every sub-expression. The type checking/inference procedure turns into a type equation solver.

Let us first present informally how we would proceed manually to infer the type of some programs.

Let us consider an example: we want to type
the following expression:

```
((lambda (x) (+ x 3)) 5)
```

Type Inference: Example 1

We consider this equation as an AST where all variable declarations have optional type annotations which are not provided. We view this expression as if it had been provided as follows in the *L5* syntax with full type annotations:

```
((lambda ((x : Tx)) : T1  
  (+ x 3))  
5)
```

Note that in this notation, type variables are used to indicate our lack of knowledge of the actual types. Our objective is to infer the value of the type variables **T_x** and **T_1** .

In addition, we also want to type check the whole expression and verify that every node in the AST can be assigned a consistent type. Therefore, we also introduce type variables for the nodes in the expression:

- T_{app} for the overall application expression
- T_{proc} for the lambda expression
- T_+ for the $+$ application expression

Given these 5 type variables: T_x , T_1 , T_{app} , T_{proc} and T_+ , we derive type constraints (equations) by inspecting the syntactic type of each expression and applying typing rules to each node:

1. $T_{proc} = [T_x \rightarrow T_1]$
2. $T_1 = T_+$

These two constraints are derived from the **procedure-typing rule** which we introduced in the previous lecture:

Type Inference: Example 1

Typing rule Procedure:

For every: type environment $_Tenv$,
variables $_x1, \dots, _xn$, $n \geq 0$
expressions $_e1, \dots, _em$, $m \geq 1$, and
type expressions $_S1, \dots, _Sn, _U1, \dots, _Um$:

Procedure with parameters ($n > 0$):

If $_Tenv \vdash \{ _x1 : _S1, \dots, _xn : _Sn \}$
| $_ei : _Ui$ for all $i = 1..m$,
Then $_Tenv \vdash (\lambda (_x1 \dots _xn) _e1 \dots _em) :$
 $[_S1 * \dots * _Sn \rightarrow _Um]$

Parameter-less Procedure ($n = 0$):

If $_Tenv \vdash _ei : _Ui$ for all $i=1..m$,
Then $_Tenv \vdash (\lambda () _e1 \dots _em) : [Empty \rightarrow _Um]$

In our case, we consider the procedure
`(lambda ((x : Tx)) : T1 (+ x 3))`
and we have named the type of the application
`(+ x 3) T+.`

The procedure typing rule mandates that:

- The type of the procedure be of the form $[T_1 \times \dots \times T_n \rightarrow T]$ where T_i are the types of the formal parameters and T the return type.
- The type of all the expressions in the body be checked.
- The return type of the procedure be the same as the type of the last expression in the body.

In general, we derive 2 constraints from each procedure node in the AST:

- One for the type of the procedure as a whole (#1 above)
- One for the return type of the procedure (#2 above)

Observe that we did not explicitly take into account the TEnv extension specified in the rule when we derived type equations. Instead, we assigned a type variable to the variable declaration (T_x for the variable x in our case) - and in the continuation of the analysis, we assume that all occurrences of x have type T_x .

This assumption relies on the implicit assumption that all the bound variables in the program have been renamed to distinct names to avoid confusion.

We then consider the application expression
and derive two more constraints:

3. $T_{\text{app}} = T_1$

4. $T_x = \text{Number}$

We derive these constraints from the
application typing rule:

Type Inference: Example 1

Typing rule Application:

For every: type environment $_TEnv$,
expressions $_f, _e1, \dots, _en$, $n \geq 0$, and
type expressions $_S1, \dots, _Sn, _S$:

Procedure with parameters ($n > 0$):

If $_TEnv \vdash _f : [_S1 * \dots * _Sn \rightarrow _S]$,
 $_TEnv \vdash _e1 : _S1, \dots, _TEnv \vdash _en : _Sn$
Then $_TEnv \vdash (_f _e1 \dots _en) : _S$

Parameter-less Procedure ($n = 0$):

If $_TEnv \vdash _f : [\text{Empty} \rightarrow _S]$
Then $_TEnv \vdash (_f) : _S$

In our case, we apply a procedure of type $T_{\text{proc}} = [T_x \rightarrow T_1]$ to the parameter 5 of type **Number**.

The rule mandates that:

- The type of the application be the same type as the return type of the procedure
 $(f \ e_1 \ \dots) = S$
- The type of the arguments be the same as the type of the formal parameters
 $(e_i = S_i)$

In general, typing an application expression creates $n + 1$ equations - one for each formal parameter e_i and one for the type of the whole application.

Finally, we consider the primitive application expression $(+ \ x \ 3)$ and derive three constraints:

- 5. $T_+ = \text{Number}$
- 6. $T_x = \text{Number}$
- 7. $T_{\text{num}3} = \text{Number}$

In general, we derive $n + 1$ type equations for every primitive application analysis: one for each of the n parameters (our primitives so far have either 1 or 2 parameters) and one for the return type of the primitive application.

Observe how primitive expressions produce extremely rich constraints - they are extremely informative. This is because primitives in our language are strongly typed - they expect a single type for their parameter and produce a single type.

Primitives in JavaScript are much less informative, because they accept variables of many types and can return many types as well.

We then solve this system of equations by systematic inspection and substitution across all equations when we find the value of a type variable.

The equations we derived are:

1. $T_{\text{proc}} = [T_x \rightarrow T_1]$

2. $T_1 = T_+$

3. $T_{\text{app}} = T_1$

4. $T_x = \text{Number}$

5. $T_+ = \text{Number}$

6. $T_x = \text{Number}$

7. $T_{\text{num3}} = \text{Number}$

- By substituting T_+ we find that $T_1 = \text{Number}$.
- By substituting T_x and T_1 we find that $T_{\text{proc}} = [\text{Number} \rightarrow \text{Number}]$.
- By substituting T_1 we find that $T_{\text{app}} = \text{Number}$.

We have thus derived the type of the overall expression. In addition, we have inferred the type of the variables T_x and T_1 and can thus provide a fully annotated version of the program:

```
((lambda ((x : Number)) : Number  
  (+ x 3))  
5)
```

Let us type the following expression - which computes the derivative of the function g with resolution dx :

```
(lambda (g dx)
  (lambda (x)
    (/ (- (g (+ x dx)) (g x))
       dx)))
```

Type Inference: Example 2

We consider this equation as an AST where all variable declarations have optional type annotations which are not provided. We view this expression as if it had been provided as follows:

```
(lambda ((g : Tg) (dx : Tdx)) : T1
  (lambda ((x : Tx)) : T2
    (/ (- (g (+ x dx)) (g x))
      dx)))
```

Our objective is to infer the value of the type variables T_g , T_{dx} , T_1 , T_x and T_2 .

In addition, we also want to type check the whole expression and verify that every node in the AST can be assigned a consistent type.

Type Inference: Example 2

Therefore, we also introduce type variables for the nodes in the expression:

1. T_{proc1} for the `lambda` in line 1
2. T_{proc2} for the `lambda` in line 2
3. $T_{/}$ for the `/` application
4. T_{-} for the `-` application
5. T_{g1} for the 1st `g` application
6. T_{+} for the `+` application
7. T_{g2} for the 2nd `g` application

This is how the AST looks with type variables in parentheses near relevant nodes: **Mermaid AST**

Given these 12 type variables: T_g , T_{dx} , T_1 , T_x , T_2 and T_{proc1} , T_{proc2} , $T_/\text{}$, T_- , T_{g1} , T_+ , T_{g2} , we derive type constraints (equations) by inspecting the syntactic type of each expression and applying typing rules to each node:

Type Inference: Example 2

- $T_{\text{proc1}} = [T_g \times T_{\text{dx}} \rightarrow T_1]$
- $T_1 = T_{\text{proc2}}$
- $T_{\text{proc2}} = [T_x \rightarrow T_2]$
- $T_2 = T_/_$
- $T_/_ = \text{Number}$
- $T_- = \text{Number}$
- $T_{\text{dx}} = \text{Number}$
- $T_- = \text{Number}$
- $T_{\text{g1}} = \text{Number}$
- $T_{\text{g2}} = \text{Number}$
- $T_g = [T_+ \rightarrow T_{\text{g1}}]$
- $T_+ = T_+$
- $T_+ = \text{Number}$
- $T_x = \text{Number}$
- $T_{\text{dx}} = \text{Number}$
- $T_g = [T_x \rightarrow T_{\text{g2}}]$
- $T_x = T_x$

We then solve this system of equations by systematic inspection and substitution across all equations when we find the value of a type variable.

By substituting T_{g2} and T_x with their values, we find:

$$T_g = [\text{Number} \rightarrow \text{Number}]$$

By substituting T_1 - we get $T_2 = \text{Number}$, then

$T_{\text{proc2}} = [\text{Number} \rightarrow \text{Number}]$, then

$T_1 = [\text{Number} \rightarrow \text{Number}]$ and eventually:

$$T_{\text{proc1}} = [[\text{Number} \rightarrow \text{Number}] \times \text{Number} \rightarrow [\text{Number} \rightarrow \text{Number}]]$$

This is the expected type for the derivative function, which given a numeric function and a resolution, returns a new numeric function.

In addition, we can provide the fully annotated version of the program:

```
(lambda ((g : (number -> number)) (dx : number)) : (number -> number)
  (lambda ((x : number)) : number
    (/ (- (g (+ x dx)) (g x))
       dx)))
```

Let us type this expression:

```
(let ((x 1))  
  (lambda (f y)  
    (f (+ x y)))))
```

We consider this equation as an AST where all variable declarations have optional type annotations which are not provided. We view this expression as if it had been provided as follows:

```
(let (((x : Tx) 1))  
  (lambda ((f : Tf) (y : Ty)) : Tres  
    (f (+ x y))))
```

Our objective is to infer the value of the type variables T_x , T_f , T_y and T_{res} . In addition, we also want to type check the whole expression and verify that every node in the AST can be assigned a consistent type.

Therefore, we also introduce type variables for the nodes in the expression:

- T_{let} for the whole **let** expression
- T_{proc} for the **lambda** expression
- T_{app} for the **f** application expression
- T_+ for the **+** application expression

Given these 8 type variables: T_x , T_f , T_y , T_{res} , T_{let} , T_{proc} , T_{app} , T_+ , we derive type constraints (equations) by inspecting the syntactic type of each expression and applying typing rules to each node:

1. $T_x = \text{Number}$
2. $T_{let} = T_{proc}$

Type Inference: Example 3

We use here the **let**-typing rule, which combines the type constraints of procedure definition and application:

Typing rule Let:

```
For every: type environment _TEnv,  
           variables _x1, ..., _xn, n >= 0  
           expressions _e1, ..., _em, m >= 1, and  
           type expressions _S1, ..., _Sn, _U1, ..., _Um :  
  
If    _TEnv o {_x1 : _S1, ..., _xn : _Sn }  
    |- _bi : _Ui for all i = 1..m  
    _TEnv |- _ei : _Si for all i = 1..n  
Then _TEnv |-  
    (let ((_x1 _e1) ... (_xn _en)) _b1 ... _bm) : _Um
```

In general, applying the **let**-typing rule yields $n + 1$ constraints, one for each binding in the **let**, and one for the type of the body being equal to the type of the whole **let**-expression.

- 3. $T_{\text{res}} = T_{\text{app}}$
- 4. $T_{\text{proc}} = [T_f \times T_y \rightarrow T_{\text{app}}]$
- 5. $T_f = [T_+ \rightarrow T_{\text{app}}]$
- 6. $T_x = \text{Number}$
- 7. $T_y = \text{Number}$
- 8. $T_+ = \text{Number}$

We then solve this system of equations by systematic inspection and substitution across all equations when we find the value of a type variable.

By substituting T_+ with its value, we find:

$$T_f = [\mathbf{Number} \rightarrow T_{app}]$$

Then, by substituting T_f and T_y by their value:

$$T_{proc} = [[\mathbf{Number} \rightarrow T_{app}] \times \mathbf{Number} \rightarrow T_{app}]$$

We eventually infer that

$$T_{\text{let}} = [[\text{Number} \rightarrow T_{\text{app}}] \times \text{Number} \rightarrow T_{\text{app}}]$$

We also have inferred the type of all the variables which appear in the annotations and can fill the annotations as follows:

```
(let (((x : number) 1))  
  (lambda ((f : (number -> Tapp)) (y : number)) : Tapp  
    (f (+ x y))))
```

We observe that the resulting type for T_{let} includes type variables. This is because this expression is polymorphic - we can compute this program for any type T_{app} and ensure that no typing error will be met.

We want to type the procedure
(**lambda** (f x) (f x x))
using type equations.

We consider this equation as an AST where all variable declarations have optional type annotations which are not provided. We view this expression as if it had been provided as follows:

```
(lambda ((f : Tf) (x : Tx)) : Tres (f x x))
```

Our objective is to infer the value of the type variables T_f and T_x . In addition, we also want to type check the whole expression and verify that every node in the AST can be assigned a consistent type.

Therefore, we also introduce type variables for the application node $(f\ x\ x)$ (T_{app}) and for the overall procedure

$(\text{lambda } (f\ x) (f\ x\ x))$ (T_{proc}).

Given these 4 type variables: T_f , T_x , T_{app} , T_{proc} ,
we derive type constraints (equations) by
inspecting the syntactic type of each expression
and applying typing rules to each node:

1. $T_f = [T_x \times T_x \rightarrow T_{app}]$
2. $T_{proc} = [T_f \times T_x \rightarrow T_{res}]$
3. $T_{res} = T_{app}$

Crucially, the same variables appear in the two equations. Our objective is to find a solution which assigns a value to the variables to make the whole system consistent. Alternatively, if there is a problem in the type of the expressions, we must detect the conflict.

Our solution consists of inspecting each equation, and to try to make the two sides of the equation equal by applying consistent substitution on both sides (this approach is quite similar to the way we solve algebraic equations).

In our case, the solution is provided by replacing T_f by its value in the right-hand side of the second equation, yielding:

$$T_{\text{proc}} = [[T_x \times T_x \rightarrow T_{\text{app}}] \times T_x \rightarrow T_{\text{app}}]$$

In addition, we have inferred the required types for all variables in the expression, and can produce the fully annotated version:

```
(lambda ((f : (Tx * Tx -> Tapp)) (x : Tx))  
      : ((Tx * Tx -> Tapp) * Tx -> Tapp)  
      (f x x))
```

We observe that the resulting expression still contains type variables (T_x and T_{app}). This is OK

- it means the procedure we have typed is polymorphic - it can work on any type pairs (T_x , T_{app}) and still be executed without leading to a typing error.

Let us now consider a case where type inference fails:

```
(lambda (x) (x x))
```

The annotated AST is:

```
(lambda ((x : Tx)) : T1 (x x))
```

The additional type variables are:

- T_{proc} for the whole **lambda** expression
- T_{app} for the application expression

We derive type equations:

1. $T_{\text{proc}} = [T_x \rightarrow T_1]$
2. $T_1 = T_{\text{app}}$
3. $T_x = [T_x \rightarrow T_{\text{app}}]$
4. $T_x = T_x$

There is no solution to this system of equations
- because we cannot solve the constraint:

$$T_x = [T_x \rightarrow T_{app}]$$

Let us now present the algorithm which performs type inference by constructing and solving type equations.

Type equation solvers use the unification algorithm for unifying type expressions and producing a consistent substitution of type variables which makes all equations equal.

The method has four stages:

1. Rename bound variables in the expression.
2. Assign type variables to all sub-expressions.
3. Construct type equations.
4. Solve the equations.

In order to describe the process of type equations more formal, we introduce the definition of type substitutions and unifiers. The notions of substitutions and renaming we will use are identical to those we introduced when describing the operational semantics of the language. The notion of unifier builds on those.

Definition: Type Substitution

A type-substitution s is a mapping from a finite set of type variables to a finite set of type expressions, such that $s(T)$ does not include T . A type-binding is a pair $\langle T; s(T) \rangle$.

Substitutions are written using set notation:

$$\{T_1 = \mathbf{Number}, T_2 = [[\mathbf{Number} \rightarrow T_3] \rightarrow T_3]\}$$

This substitution:

$$\{T_1 = \mathbf{Number}, T_2 = [[\mathbf{Number} \rightarrow T_3] \rightarrow T_2]\}$$

is an illegal substitution because T_2 occurs in $s(T_2)$.

Definition: Substitution Application

The application of a type-substitution s to a type expression T , denoted $T \circ s$ (or just Ts), consistently replaces all occurrences of type variables T_i in T by their mapped type expressions $s(T_i)$. The replacement is simultaneous.

For example:

$$\begin{aligned} & [[T_1 \rightarrow T_2] \rightarrow T_2] \circ \{T_1 = \text{Boolean}, T_2 = [T_3 \rightarrow T_3]\} \\ &= [[\text{Boolean} \rightarrow [T_3 \rightarrow T_3]] \rightarrow [T_3 \rightarrow T_3]] \end{aligned}$$

Definition: Type Instance, More General Relation

We say that a type expression T' is an instance of a type expression T , if there is a type substitution s such that $T \circ s = T'$.

T is more general than T' , if T' is an instance of T .

The following type expressions are instances of $[T \rightarrow T]$:

- $[\text{Number} \rightarrow \text{Number}] = [T \rightarrow T] \circ \{T = \text{Number}\}$
- $[\text{Symbol} \rightarrow \text{Symbol}] = [T \rightarrow T] \circ \{T = \text{Symbol}\}$
- $[[\text{Number} \rightarrow \text{Number}] \rightarrow [\text{Number} \rightarrow \text{Number}]] = [T \rightarrow T] \circ \{T = [\text{Number} \rightarrow \text{Number}]\}$
- $[[\text{Number} \rightarrow T_1] \rightarrow [\text{Number} \rightarrow T_1]] = [T \rightarrow T] \circ \{T = [\text{Number} \rightarrow T_1]\}$

Definition: Combination (composition) of Type Substitutions

The combination of type-substitutions s and s' , denoted $s \circ s'$, is an operation that results in a type-substitution, or fails.

Definition: Combination (composition) of Type Substitutions

It is defined by:

1. s' is applied to the type-expressions of s , *i.e.*, for every variable T' for which $s'(T')$ is defined, occurrences of T' in type expressions in s are replaced by $s'(T')$.
2. A variable T in s' , for which $s(T)$ is defined, is removed from the domain of s' , *i.e.*, $s'(T)$ is not defined on it anymore.

Definition: Combination (composition) of Type Substitutions

3. The modified s' is added to s .
4. Identity bindings, *i.e.*, $s(T) = T$, are removed.
5. If for some variable, $(s \circ s')(T)$ includes T , the combination fails.

For example:

$$\{T_1 = \text{Number}, T_2 = [[\text{Number} \rightarrow T_3] \rightarrow T_3]\} \circ \\ \{T_3 = \text{Boolean}, T_1 = [T_2 \rightarrow T_2]\}$$

=

$$\{T_1 = \text{Number}, \\ T_2 = [[\text{Number} \rightarrow \text{Boolean}] \rightarrow \text{Boolean}], \\ T_3 = \text{Boolean}\}$$

Definition: Renaming of Type Variables

Renaming is the operation of consistent renaming of type variables within a type expression, by new type symbols, that do not occur in the type expression.

Renamed type expressions are equivalent:

$$[[T_1 \rightarrow T_2] \times T_1 \rightarrow T_2] \sim [[S_1 \rightarrow T_2] \times S_1 \rightarrow T_2]$$

$$[[T_1 \rightarrow T_2] \times T_1 \rightarrow T_2] \sim [[S_1 \rightarrow S_2] \times S_1 \rightarrow S_2]$$

The variables in the substituting expressions should be new. For example, the following renamings of $[[T_1 \rightarrow T_2] \times T_1 \rightarrow T_2]$ are illegal:

- $[[T_1 \rightarrow T_2] \times S_2 \rightarrow T_2] - T_1$ is not consistently replaced, T_1 and S_2 are not kept together.
- $[[T_2 \rightarrow T_2] \times T_2 \rightarrow T_2] - T_2$ is reused to replace T_1

Definition: Unification of Type Expressions

Unification is an operation that makes type expressions identical by application of a type substitution to both expressions. When such a substitution can be found, it is called a **unifier** of the two type expressions.

For example:

$$\begin{aligned} & [S \times [\text{Number} \rightarrow S] \rightarrow S] \circ \\ \{S = \text{Pair}(T_1), T_2 = [\text{Number} \rightarrow \text{Pair}(T_1)], T_3 = \text{Pair}(T_1)\} = \\ & [\text{Pair}(T_1) \times T_2 \rightarrow T_3] \circ \\ \{S = \text{Pair}(T_1), T_2 = [\text{Number} \rightarrow \text{Pair}(T_1)], T_3 = \text{Pair}(T_1)\} = \\ & [\text{Pair}(T_1) \times [\text{Number} \rightarrow \text{Pair}(T_1)] \rightarrow \text{Pair}(T_1)] \end{aligned}$$

Therefore,

$\{S = \text{Pair}(T_1), T_2 = [\text{Number} \rightarrow \text{Pair}(T_1)], T_3 = \text{Pair}(T_1)$
is a **unifier** for these type expressions.

Definition: Unifier of Type Expressions

A unifier of type expressions T_1, T_2 is a type substitution s such that $T_1 \circ s = T_2 \circ s$.

The type expressions should not include common type variables! (Apply renaming, if needed.)

For example, consider the type expressions

$[S \times [\text{Number} \rightarrow S_1] \rightarrow S]$ and
 $[\text{Pair}(T_1) \times [T_1 \rightarrow T_1] \rightarrow T_2]$.

These expressions are unifiable by:

$\{S = \text{Pair}(\text{Number}),$
 $T_1 = \text{Number},$
 $S_1 = \text{Number},$
 $T_2 = \text{Pair}(\text{Number})\}$

Type Substitutions and Unifiers

For example, consider the type expressions

$$[S \times [\text{Number} \rightarrow S] \rightarrow S] \text{ and } [\text{Pair}(T_1) \times [T_1 \rightarrow T_1] \rightarrow T_2].$$

They are **not** unifiable because we would need to resolve (find a substitution which leads to the equality):

$$S = \text{Pair}(T_1) \text{ and } S = T_2 \text{ and } [\text{Number} \rightarrow S] = [T_1 \rightarrow T_1], \text{ hence } S = T_1, \\ \text{which is not compatible with } T_1 = \text{Pair}(T_1)$$

Type Substitutions and Unifiers

Definition: Most General Unifier (mgu)

Unifiable type expressions can be unified by **multiple unifiers**.

For example, the type expressions $[S \times S \rightarrow S]$ and $[\text{Pair}(T_1) \times T_2 \rightarrow T_2]$ are unifiable by the unifiers:

- $\{S = \text{Pair}(T_1), T_2 = \text{Pair}(T_1)\}$
- $\{S = \text{Pair}(\text{Number}), T_2 = \text{Pair}(\text{Number})\}$
- $\{S = \text{Pair}(\text{Boolean}), T_2 = \text{Pair}(\text{Boolean})\}$
- etc.

Definition: Most General Unifier (mgu)

The first unifier is the most general unifier (mgu), since it substitutes only the necessary type variables, without making additional assumptions about the replaced terms.

Definition: Most General Unifier (mgu)

All other unifiers are obtained from it by application of additional substitutions. The most general unifier is unique, up to consistent renaming. It is called the most general unifier (mgu) of the two type expressions.

We will use in the type inference algorithm the function `unify(TE1, TE2)` which returns the mgu of `TE1` and `TE2` if it can be found and `false` otherwise (indicating the two expressions cannot be unified).

With the unification tool at our disposal, let us run the full details of the type inference algorithm step by step:

We want to type the following expression:

```
(lambda (f g)
  (lambda (x)
    (f (+ x (g 3))))))
```

We consider it with its type annotations:

```
(lambda ((f : Tf) (g : Tg)) : T1  
  (lambda ((x : Tx)) : T2  
    (f (+ x (g 3))))))
```

Stage I: Renaming:

None needed because all declared variables already have distinct names.

Stage II: Assign type variables: Every sub expression is assigned a type variable:

Type Inference With Equations Step by Step

Expression	Variable
(lambda (f g) ...)	T_0
(lambda (x) ...)	T_1
(f (+ x (g 3)))	T_2
f	T_f
(+ x (g 3))	T_3
+	T_+
x	T_x
(g 3)	T_4
g	T_g
3	$T_{\text{num}3}$

Stage III: Construct type equations:

The typing rules of algorithm Type-derivation turn into type equations: The rules are:

1. Number, Boolean, Symbol,
Primitive-procedures: Construct equations using their types. For example, for the number 3: $T_{\text{num3}} = \mathbf{Number}$, and for the binary primitive procedure +:
 $T_{+} = [\mathbf{Number} \times \mathbf{Number} \rightarrow \mathbf{Number}]$

2. Procedure (`lambda` expressions):

Type Inference With Equations Step by Step

Typing rule Procedure:

For every: type environment $_TEnv$,
variables $_x1, \dots, _xn$, $n \geq 0$
expressions $_e1, \dots, _em$, $m \geq 1$, and
type expressions $_S1, \dots, _Sn, _U1, \dots, _Um$:

Procedure with parameters ($n > 0$):

If $_TEnv \vdash \{ _x1 : _S1, \dots, _xn : _Sn \}$
| $_ei : _Ui$ for all $i = 1..m$,
Then $_TEnv \vdash (\lambda (_x1 \dots _xn) _e1 \dots _em) :$
 $[_S1 * \dots * _Sn \rightarrow _Um]$

Parameter-less Procedure ($n = 0$):

If $_TEnv \vdash _ei : _Ui$ for all $i=1..m$,
Then $_TEnv \vdash (\lambda () _e1 \dots _em) : [Empty \rightarrow _Um]$

Extracting the type restriction on the involved sub-expressions yields:

- For `(lambda (v1 ... vn) e1 ... em)`, construct the equation:

$$T_{(\text{lambda } (v1 \dots vn) e1 \dots em)} = [T_{v1} \times \dots \times T_{vn} \rightarrow T_{em}]$$

- For `(lambda () e1 ... em)`, construct the equation:

$$T_{(\text{lambda } () e1 \dots em)} = [\text{Empty} \rightarrow T_{em}]$$

3. Application: The type-inference rule is:

Typing rule Application:

For every: type environment $_TEnv$,
expressions $_f, _e1, \dots, _en$, $n \geq 0$, and
type expressions $_S1, \dots, _Sn, _S$:

Procedure with parameters ($n > 0$):

```
If   $\_TEnv \vdash \_f : [\_S1 * \dots * \_Sn \rightarrow \_S]$ ,  
     $\_TEnv \vdash \_e1 : \_S1, \dots, \_TEnv \vdash \_en : \_Sn$   
Then  $\_TEnv \vdash (\_f \_e1 \dots \_en) : \_S$ 
```

Parameter-less Procedure ($n = 0$):

```
If   $\_TEnv \vdash \_f : [Empty \rightarrow \_S]$   
Then  $\_TEnv \vdash (\_f) : \_S$ 
```

Extracting the type restriction on the involved sub-expressions yields:

- For $(f \ e1 \ \dots \ en)$ with $n > 0$, construct the equation:

$$T_f = [T_{e1} \times \dots \times T_{en} \rightarrow T_{(f \ e1 \ \dots \ en)}]$$

- For (f) construct the equation:

$$T_f = [\text{Empty} \rightarrow T_{(f)}]$$

Note: Observe that the inference rules for Procedure and Application require type inference for all internal expressions, even though the final inferred type does not depend on their type. Why?

In the type-equations approach this requirement is achieved by constructing type equations for all sub-expressions, as described below. The algorithm constructs equations for the primitive sub-expressions and for all composite sub-expressions.

In our example:

The equations for the primitive sub-expressions are:

Expression	Equation
3	$T_{\text{num}3} = \text{Number}$
+	$T_+ = [\text{Number} \times \text{Number} \rightarrow \text{Number}]$

The equations for composite sub-expressions are:

Expression	Equation
<code>(lambda (f g) ...)</code>	$T_0 = [T_f \times T_g \rightarrow T_1]$
<code>(lambda (x) ...)</code>	$T_1 = [T_x \rightarrow T_2]$
<code>(f (+ x (g 3)))</code>	$T_f = [T_3 \rightarrow T_2]$
<code>(+ x (g 3))</code>	$T_+ = [T_x \times T_4 \rightarrow T_3]$
<code>(g 3)</code>	$T_g = [T_{\text{num}3} \rightarrow T_4]$

Stage IV: Solving the Equations:

The equations are solved by gradually producing type-substitutions for all type variables. For an expression e , the algorithm infers a type t if the final type-substitution maps its variable T_e to t .

If an expression has an inferred type then all of its sub-expressions have types as well. If the procedure fails (output is **FAIL**) then either there is a type error or the constructed type equations are too weak.

Circular type-substitution cause failure.

The solution is processed by considering the equations one by one.

The equation solving process is described by this algorithm:

Type Inference With Equations Step by Step

Input: A set of type equations.

Output: A type substitution or FAIL.

Initialization:

1. substitution := { }
2. Order the set of input equations in any sequential order.
3. equation := $te_1 = te_2$, the first equation.

Type Inference With Equations Step by Step

Loop:

1. Apply the current substitution to the equation:

Let $te1s := te1 \circ substitution$

$te2s := te2 \circ substitution$

$equation := te1s = te2s$

2. If $te1s$ and $te2s$ are atomic types:

If $te1s \neq te2s$:

sub = FAIL

otherwise:

Do nothing.

3. Without loss of generality:

If $te1s = T$, i.e., a type variable,
and $te1s \neq te2s$:

$substitution := substitution \circ \{T = te2s\}$

That is, apply the equation to substitution,
and add the equation to the substitution.
If the application fails (circular mapping),
 $substitution := FAIL$.

4. If `te1s` and `te2s` are composite types:
 - If they have the same type constructor:
 - Split `te1s` and `te2s` into component type expressions, create equations for corresponding components, and add the new equations to the pool of equations.
 - otherwise:
 - `substitution := FAIL`

Type Inference With Equations Step by Step

5. Without loss of generality:
 If `te1s` is an atomic type
 and `te2s` is a composite type:
 `substitution := FAIL`
6. If there is a next equation:
 `equation := next(equations)`

Until `substitution = FAIL` or no more equations

Return: `substitution`

This algorithm computes the unifier of all the equations into a single consistent type substitution.

In other words, each time we process an equation $T_{e1} = T_{e2}$, we make the two sides equal by finding their unifier and then apply the resulting unifier to the remaining equations and continue the process.

The constraints flow from equation to equation because we re-use the same substitution across equations and, thus, propagate information from one equation to the next.

Let us continue our example applying this algorithm to solve the equations:

Type Inference With Equations Step by Step

Equations	Substitution
1. $T_\theta = [T_f \times T_g \rightarrow T_1]$	
2. $T_1 = [T_x \rightarrow T_2]$	
3. $T_f = [T_3 \rightarrow T_2]$	
4. $T_+ = [T_x \times T_4 \rightarrow T_3]$	
5. $T_g = [T_{\text{num3}} \rightarrow T_4]$	
6. $T_{\text{num3}} = N$	
7. $T_+ = [N \times N \rightarrow N]$	

Type Inference With Equations Step by Step

Equations	Substitution
2. $T_1 = [T_x \rightarrow T_2]$ 3. $T_f = [T_3 \rightarrow T_2]$ 4. $T_+ = [T_x \times T_4 \rightarrow T_3]$ 5. $T_g = [T_{\text{num3}} \rightarrow T_4]$ 6. $T_{\text{num3}} = N$ 7. $T_+ = [N \times N \rightarrow N]$	$T_\theta = [T_f \times T_g \rightarrow T_1]$

Type Inference With Equations Step by Step

Equations	Substitution
3. $T_f = [T_3 \rightarrow T_2]$ 4. $T_+ = [T_x \times T_4 \rightarrow T_3]$ 5. $T_g = [T_{\text{num3}} \rightarrow T_4]$ 6. $T_{\text{num3}} = N$ 7. $T_+ = [N \times N \rightarrow N]$	$T_0 = [T_f \times T_g \rightarrow [T_x \rightarrow T_2]]$ $T_1 = [T_x \rightarrow T_2]$

Type Inference With Equations Step by Step

Equations	Substitution
4. $T_+ = [T_x \times T_4 \rightarrow T_3]$	$T_0 = [[T_3 \rightarrow T_2] \times T_g \rightarrow [T_x \rightarrow T_2]]$
5. $T_g = [T_{\text{num3}} \rightarrow T_4]$	$T_1 = [T_x \rightarrow T_2]$
6. $T_{\text{num3}} = N$	$T_f = [T_3 \rightarrow T_2]$
7. $T_+ = [N \times N \rightarrow N]$	

Type Inference With Equations Step by Step

Equations	Substitution
5. $T_g = [T_{\text{num3}} \rightarrow T_4]$	$T_\theta = [[T_3 \rightarrow T_2] \times T_g \rightarrow [T_x \rightarrow T_2]]$
6. $T_{\text{num3}} = N$	$T_1 = [T_x \rightarrow T_2]$
7. $T_+ = [N \times N \rightarrow N]$	$T_f = [T_3 \rightarrow T_2]$
	$T_+ = [T_x \times T_4 \rightarrow T_3]$

Type Inference With Equations Step by Step

Equations	Substitution
6. $T_{\text{num3}} = N$ 7. $T_+ = [N \times N \rightarrow N]$	$T_0 = [[T_3 \rightarrow T_2] \times [T_{\text{num3}} \rightarrow T_4] \rightarrow [T_x \rightarrow T_2]]$ $T_1 = [T_x \rightarrow T_2]$ $T_f = [T_3 \rightarrow T_2]$ $T_+ = [T_x \times T_4 \rightarrow T_3]$ $T_g = [T_{\text{num3}} \rightarrow T_4]$

Type Inference With Equations Step by Step

Equations	Substitution
7. $T_+ = [N \times N \rightarrow N]$	$T_\theta = [[T_3 \rightarrow T_2] \times [N \rightarrow T_4] \rightarrow [T_x \rightarrow T_2]]$ $T_1 = [T_x \rightarrow T_2]$ $T_f = [T_3 \rightarrow T_2]$ $T_+ = [T_x \times T_4 \rightarrow T_3]$ $T_g = [N \rightarrow T_4]$ $T_{\text{num3}} = N$

Type Inference With Equations Step by Step

Equations	Substitution
8. $T_x = N$	$T_\theta = [(T_3 \rightarrow T_2) \times [N \rightarrow T_4] \rightarrow [T_x \rightarrow T_2]]$
9. $T_4 = N$	$T_1 = [T_x \rightarrow T_2]$
10. $T_3 = N$	$T_f = [T_3 \rightarrow T_2]$
	$T_g = [N \rightarrow T_4]$
	$T_{\text{num3}} = N$

Type Inference With Equations Step by Step

Equations	Substitution
	$T_0 = [[N \rightarrow T_2] \times [N \rightarrow N] \rightarrow [N \rightarrow T_2]]$ $T_1 = [N \rightarrow T_2]$ $T_f = [N \rightarrow T_2]$ $T_g = [N \rightarrow N]$ $T_{\text{num3}} = N$ $T_x = N$ $T_4 = N$ $T_3 = N$

On the basis of this substitution, we can return the fully annotated expression:

Type Inference With Equations Step by Step

```
(lambda ((f : Tf) (g : Tg)) : T1  
  (lambda ((x : Tx)) : T2  
    (f (+ x (g 3))))))
```

becomes:

```
(lambda ((f : (number -> T2))  
        (g : (number -> number))) : (number -> T2)  
  (lambda ((x : number)) : T2  
    (f (+ x (g 3))))))
```