

Lecture 26 – Type Inference (2)

COSE212: Programming Languages

Jihyeok Park



2024 Fall

- **Type inference** is the process of automatically inferring the types of expressions.
- We have seen three examples to learn how the type inference works.

```
/* RFAE */ def sum(x) = if (x < 1) 0 else x + sum(x - 1); sum
```

```
/* FAE */ val app = n => f => f(n); app(42)(x => x)
```

```
/* FAE */ val id = x => x; val n = id(42); val b = id(true); b
```

- In this lecture, let's learn the details of the type inference algorithm.
- **TIFAE** – TRFAE with **type inference**.
 - **Type Checker** and **Typing Rules** with **Type Inference**
 - Interpreter and Natural Semantics

1. Type Checker and Typing Rules with Type Inference
 - Solutions for Type Constraints
 - Numbers
 - Additions
 - Conditionals
 - Immutable Variable Definitions and Identifier Lookup
 - Function Definitions
 - Recursive Function Definitions
 - Function Applications
2. Type Unification
 - Type Resolving
 - Occurrence Checking
 - Type Unification
3. Type Inference with Let-Polymorphism
 - Type Generalization
 - Type Instantiation

1. Type Checker and Typing Rules with Type Inference
 - Solutions for Type Constraints
 - Numbers
 - Additions
 - Conditionals
 - Immutable Variable Definitions and Identifier Lookup
 - Function Definitions
 - Recursive Function Definitions
 - Function Applications
2. Type Unification
 - Type Resolving
 - Occurrence Checking
 - Type Unification
3. Type Inference with Let-Polymorphism
 - Type Generalization
 - Type Instantiation

Let's ① design **typing rules** of TIFAE to define when an expression is well-typed in the form of:

$$\Gamma \vdash e : \tau$$

and ② implement a **type checker** in Scala according to typing rules:

```
def typeCheck(expr: Expr, tenv: TypeEnv): Type = ???
```

The type checker returns the **type** of e if it is well-typed, or rejects it and throws a **type error** otherwise.

We will keep track of the **variable types** using a **type environment** Γ as a mapping from variable names to their types.

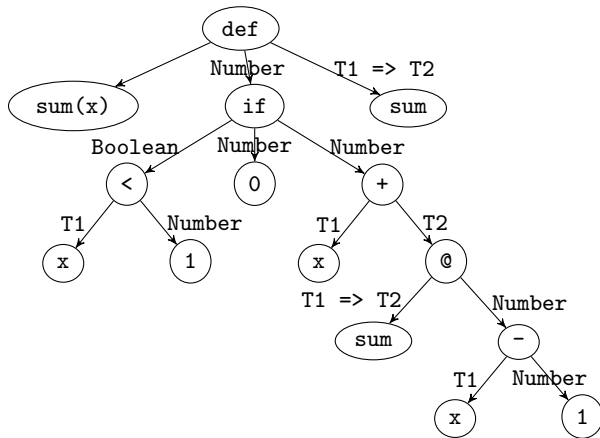
Type Environments $\Gamma \in \Gamma = \mathbb{X} \xrightarrow{\text{fin}} \mathbb{T}$ (TypeEnv)

```
type TypeEnv = Map[String, Type]
```

Recall: Example 1 – sum

In addition, we need to keep track of the **solution** for **type constraints** over **type variables** to infer the types of expressions.

```
/* RFAE */ def sum(x) = if (x < 1) 0 else x + sum(x - 1); sum
```



Type Environment

X	T
x	$T1$
sum	$T1 \Rightarrow T2$

Solution

X_α	T
$T1$	$Number$
$T2$	$Number$

A **solution** is a mapping from **type variables** to **types** or **•**.

Types $\mathbb{T} \ni \tau ::= \text{num} \mid \text{bool} \mid \tau \rightarrow \tau \mid \alpha \quad (\text{Type})$

Solutions $\psi \in \Psi = \mathbb{X}_\alpha \xrightarrow{\text{fin}} (\mathbb{T} \uplus \{\bullet\}) \quad (\text{Solution})$

Type Variables $\alpha \in \mathbb{X}_\alpha \quad (\text{Int})$

```
type Solution = Map[Int, Option[Type]]
```

Note that **•** (None) represents a **not yet solved (free)** type variable.

Now, we have new forms of **type checker** and **typing rules**.

```
def typeCheck(expr: Expr, tenv: TypeEnv, sol: Solution): (Type, Solution) = ???
```

$$\Gamma, \psi \vdash e : \tau, \psi$$

Similar to the memory passing in MFAE for mutation, we will pass the solution ψ and update it during type checking.

```
def typeCheck(e: Expr, tenv: TypeEnv, sol: Solution): (Type, Solution) = e match
  ...
  case Num(n) => (NumT, sol)
```

$$\boxed{\Gamma, \psi \vdash e : \tau, \psi}$$

$$\tau\text{-Num} \frac{}{\Gamma, \psi \vdash n : \text{num}, \psi}$$


```
def typeCheck(e: Expr, tenv: TypeEnv, sol: Solution): (Type, Solution) = e match
  ...
  case Add(l, r) =>
    val (lty, sol1) = typeCheck(l, tenv, sol)
    val (rty, sol2) = typeCheck(r, tenv, sol1)
    val sol3 = unify(lty, NumT, sol2)
    val sol4 = unify(rty, NumT, sol3)
    (NumT, sol4)
```

$$\boxed{\Gamma, \psi \vdash e : \tau, \psi}$$

$$\tau\text{-Add} \frac{\Gamma, \psi_0 \vdash e_1 : \tau_1, \psi_1 \quad \Gamma, \psi_1 \vdash e_2 : \tau_2, \psi_2 \quad \text{unify}(\tau_1, \text{num}, \psi_2) = \psi_3 \quad \text{unify}(\tau_2, \text{num}, \psi_3) = \psi_4}{\Gamma, \psi_0 \vdash e_1 + e_2 : \text{num}, \psi_4}$$

The unify function that takes two types must be the same and updates the given solution. We will see how it works later.

```

def typeCheck(e: Expr, tenv: TypeEnv, sol: Solution): (Type, Solution) = e match
...
case If(c, t, e) =>
  val (cty, sol1) = typeCheck(c, tenv, sol)
  val (tty, sol2) = typeCheck(t, tenv, sol1)
  val (ety, sol3) = typeCheck(e, tenv, sol2)
  val sol4 = unify(cty, BoolT, sol3)
  val sol5 = unify(tty, ety, sol4)
  (tty, sol5)

```

$$\boxed{\Gamma, \psi \vdash e : \tau, \psi}$$

$$\tau\text{-If} \frac{\Gamma, \psi \vdash e_c : \tau_c, \psi_c \quad \Gamma, \psi_c \vdash e_t : \tau_t, \psi_t \quad \Gamma, \psi_t \vdash e_e : \tau_e, \psi_e \quad \text{unify}(\tau_c, \text{bool}, \psi_e) = \psi' \quad \text{unify}(\tau_t, \tau_e, \psi') = \psi''}{\Gamma, \psi \vdash \text{if } (e_c) e_t \text{ else } e_e : \tau_t, \psi''}$$

```

def typeCheck(e: Expr, tenv: TypeEnv, sol: Solution): (Type, Solution) = e match
  ...

case Val(x, e, b) =>
  val (ety, sol1) = typeCheck(e, tenv, sol)
  typeCheck(b, tenv + (x -> ety), sol1)

case Id(x) =>
  val ty = tenv.getOrElse(x, error(s"free identifier: $x"))
  (ty, sol)

```

$$\boxed{\Gamma, \psi \vdash e : \tau, \psi}$$

$$\tau\text{-Val} \frac{\Gamma, \psi_0 \vdash e_1 : \tau_1, \psi_1 \quad \Gamma[x : \tau_1], \psi_1 \vdash e_2 : \tau_2, \psi_2}{\Gamma, \psi_0 \vdash \text{val } x = e_1; e_2 : \tau_2, \psi_2}$$

$$\tau\text{-Id} \frac{x \in \text{Domain}(\Gamma)}{\Gamma, \psi \vdash x : \Gamma(x), \psi}$$

```
def typeCheck(e: Expr, tenv: TypeEnv, sol: Solution): (Type, Solution) = e match
...
case Fun(p, b) =>
  val (pty, sol1) = newTypeVar(sol)
  val (rty, sol2) = typeCheck(b, tenv + (p -> pty), sol1)
  (ArrowT(pty, rty), sol2)
```

$$\boxed{\Gamma, \psi \vdash e : \tau, \psi}$$

$$\tau\text{-Fun} \frac{\alpha_p \notin \psi \quad \Gamma[x : \alpha_p], \psi[\alpha_p \mapsto \bullet] \vdash e : \tau, \psi'}{\Gamma, \psi \vdash \lambda x. e : \alpha_p \rightarrow \tau, \psi'}$$

We need to introduce a **new type variable** α_p for the parameter x .

```
def typeCheck(e: Expr, tenv: TypeEnv, sol: Solution): (Type, Solution) = e match
...
case Rec(f, p, b, s) =>
  val (pty, sol1) = newTypeVar(sol)
  val (rty, sol2) = newTypeVar(sol1)
  val fty = ArrowT(pty, rty)
  val tenv1 = tenv + (f -> fty)
  val tenv2 = tenv1 + (p -> pty)
  val (bty, sol3) = typeCheck(b, tenv2, sol2)
  val sol4 = unify(bty, rty, sol3)
  typeCheck(s, tenv1, sol4)
```

$$\boxed{\Gamma, \psi \vdash e : \tau, \psi}$$

$$\tau\text{-Rec} \frac{\begin{array}{l} \alpha_p, \alpha_r \notin \psi \quad \alpha_p \neq \alpha_r \quad \Gamma_1 = \Gamma[x_f : (\alpha_p \rightarrow \alpha_r)] \\ \Gamma_2 = \Gamma_1[x_p : \alpha_p] \quad \Gamma_2, \psi[\alpha_p \mapsto \bullet, \alpha_r \mapsto \bullet] \vdash e_b : \tau_b, \psi_b \\ \text{unify}(\tau_b, \alpha_r, \psi_b) = \psi_r \quad \Gamma_1, \psi_r \vdash e_s : \tau_s, \psi_s \end{array}}{\Gamma, \psi \vdash \text{def } x_f(x_p) = e_b; e_s : \tau_s, \psi_s}$$

```
def typeCheck(e: Expr, tenv: TypeEnv, sol: Solution): (Type, Solution) = e match
...
case App(f, a) =>
  val (fty, sol1) = typeCheck(f, tenv, sol)
  val (aty, sol2) = typeCheck(a, tenv, sol1)
  val (rty, sol3) = newTypeVar(sol2)
  val sol4 = unify(ArrowT(aty, rty), fty, sol3)
  (rty, sol4)
```

$$\boxed{\Gamma, \psi \vdash e : \tau, \psi}$$

$$\tau\text{-App} \frac{\Gamma, \psi \vdash e_f : \tau_f, \psi_f \quad \Gamma, \psi_f \vdash e_a : \tau_a, \psi_a \quad \alpha_r \notin \psi_a \quad \text{unify}(\tau_a \rightarrow \alpha_r, \tau_f, \psi_a[\alpha_r \mapsto \bullet]) = \psi'}{\Gamma, \psi \vdash e_f(e_a) : \alpha_r, \psi'}$$

1. Type Checker and Typing Rules with Type Inference
 - Solutions for Type Constraints
 - Numbers
 - Additions
 - Conditionals
 - Immutable Variable Definitions and Identifier Lookup
 - Function Definitions
 - Recursive Function Definitions
 - Function Applications
2. Type Unification
 - Type Resolving
 - Occurrence Checking
 - Type Unification
3. Type Inference with Let-Polymorphism
 - Type Generalization
 - Type Instantiation

Definition (Type Unification)

Type unification is the process of updating a solution to make two types equal. If the types are not unifiable, then this process fails and throws an exception.

$$\text{unify} : (\mathbb{T} \times \mathbb{T} \times \Psi) \rightarrow \Psi$$

For example, if we unify a type variable α and the number type `num`, the solution $[\alpha \mapsto \bullet]$ is updated to $[\alpha \mapsto \text{num}]$.

$$\text{unify}(\alpha, \text{num}, \emptyset) = [\alpha \mapsto \text{num}]$$

Before, we define the type unification, we need to define the **type resolving** and **occurrence checking** functions.

- 1 **Type resolving** is the process of recursively resolving a type variable to its representative type to deal with the **type aliasing**.
- 2 **Occurrence checking** is the process of checking whether a type variable occurs in a type to detect **recursive types**.

To understand why we need the **type resolving** function, let's consider the following example:

$$\text{unify}(\alpha_1, \text{num}, \psi_1) = \psi_2$$

Solution	
X_α	T
α_1	α_2
α_2	α_3
α_3	•

 $\psi_1 =$

Solution	
X_α	T
α_1	num
α_2	α_3
α_3	•

 $\psi_2 =$

If we update α_1 to num in the solution ψ_2 , it misses the information that α_2 and α_3 are also num.

To understand why we need the **type resolving** function, let's consider the following example:

$$\text{unify}(\alpha_1, \text{num}, \psi_1) = \psi_2$$

Solution		Solution	
X_α	T	X_α	T
α_1	α_2	α_1	α_2
α_2	α_3	α_2	α_3
α_3	•	α_3	num

If we directly update α_1 to num in the solution ψ_2 , it misses the information that α_2 and α_3 are also num.

Instead, we need to **resolve** the type variable α_1 to find its **representative type** (i.e., α_3) and unify it with num to deal with the **type aliasing**.

We can define the **type resolving** function as follows:

$$\text{resolve} : (\mathbb{T} \times \Psi) \rightarrow \mathbb{T}$$

$$\text{resolve}(\tau, \psi) = \begin{cases} \text{resolve}(\tau', \psi) & \text{if } \tau = \alpha \wedge \psi(\alpha) = \tau' \\ \tau & \text{otherwise} \end{cases}$$

and implement it in Scala as follows:

```
def resolve(ty: Type, sol: Solution): Type = ty match
  case VarT(k) => sol(k) match
    case Some(ty) => resolve(ty, sol)
    case None => ty
  case _ => ty
```

Let's understand why we need the **occurrence checking** function:

$$\text{unify}(\alpha_1, \text{num} \rightarrow \alpha_1, \psi) = \psi'$$

Can we unify α_1 and $\text{num} \rightarrow \alpha_1$? **No!** because it requires **recursive types** not supported in our type system.

Let's define the **occurrence checking** function to detect type constraints that require recursive types

$$\text{occur} : (\mathbb{X}_\alpha \times \mathbb{T} \times \Psi) \rightarrow \text{bool}$$

$$\text{occur}(\alpha, \tau, \psi) = \begin{cases} \text{true} & \text{if } \tau = \alpha \\ \text{occur}(\alpha, \tau_p, \psi) \vee \text{occur}(\alpha, \tau_r, \psi) & \text{if } \tau = (\tau_p \rightarrow \tau_r) \\ \text{false} & \text{otherwise} \end{cases}$$

and implement it in Scala as follows:

```
def occurs(k: Int, ty: Type, sol: Solution): Boolean = resolve(ty, sol) match
  case VarT(l) => k == l
  case ArrowT(pty, rty) => occurs(k, pty, sol) || occurs(k, rty, sol)
  case _ => false
```

Using the **type resolving** and **occurrence checking** functions, we could define the **type unification** as a partial function:

$$\text{unify} : (\mathbb{T} \times \mathbb{T} \times \Psi) \rightarrow \Psi$$

$$\text{unify}(\tau_1, \tau_2, \psi) =$$

{

1

2

3

Using the **type resolving** and **occurrence checking** functions, we could define the **type unification** as a partial function:

$$\text{unify} : (\mathbb{T} \times \mathbb{T} \times \Psi) \rightarrow \Psi$$

$$\text{unify}(\tau_1, \tau_2, \psi) = \left\{ \begin{array}{l} \\ \\ \\ \\ \\ \end{array} \right.$$

where $\tau'_1 = \text{resolve}(\tau_1, \psi)$ and $\tau'_2 = \text{resolve}(\tau_2, \psi)$.

- 1 First, it resolves the types τ_1 and τ_2 with the current solution ψ into τ'_1 and τ'_2 using the **type resolving** function `resolve`.

2

3

Using the **type resolving** and **occurrence checking** functions, we could define the **type unification** as a partial function:

$$\text{unify} : (\mathbb{T} \times \mathbb{T} \times \Psi) \rightarrow \Psi$$

$$\text{unify}(\tau_1, \tau_2, \psi) = \begin{cases} \psi[\alpha \mapsto \tau_2'] & \text{if } \tau_1' = \alpha \wedge \neg \text{occur}(\alpha, \tau_2') \\ \psi[\alpha \mapsto \tau_1'] & \text{if } \tau_2' = \alpha \wedge \neg \text{occur}(\alpha, \tau_1') \end{cases}$$

where $\tau_1' = \text{resolve}(\tau_1, \psi)$ and $\tau_2' = \text{resolve}(\tau_2, \psi)$.

- 1 First, it resolves the types τ_1 and τ_2 with the current solution ψ into τ_1' and τ_2' using the **type resolving** function `resolve`.
- 2 If one of τ_1' or τ_2' is a type variable, it checks recursive types using the **occurrence checking** and updates the solution of the type variable.

3

Using the **type resolving** and **occurrence checking** functions, we could define the **type unification** as a partial function:

$$\text{unify} : (\mathbb{T} \times \mathbb{T} \times \Psi) \rightarrow \Psi$$

$$\text{unify}(\tau_1, \tau_2, \psi) = \begin{cases} \psi & \text{if } \tau'_1 = \text{num} \wedge \tau'_2 = \text{num} \\ \psi & \text{if } \tau'_1 = \text{bool} \wedge \tau'_2 = \text{bool} \\ \text{unify}(\tau_{1,r}, \tau_{2,r}, \text{unify}(\tau_{1,p}, \tau_{2,p}, \psi)) & \text{if } \tau'_1 = (\tau_{1,p} \rightarrow \tau_{1,r}) \wedge \tau'_2 = (\tau_{2,p} \rightarrow \tau_{2,r}) \\ \psi & \text{if } \tau'_1 = \alpha = \tau'_2 \\ \psi[\alpha \mapsto \tau'_2] & \text{if } \tau'_1 = \alpha \wedge \neg \text{occur}(\alpha, \tau'_2) \\ \psi[\alpha \mapsto \tau'_1] & \text{if } \tau'_2 = \alpha \wedge \neg \text{occur}(\alpha, \tau'_1) \end{cases}$$

where $\tau'_1 = \text{resolve}(\tau_1, \psi)$ and $\tau'_2 = \text{resolve}(\tau_2, \psi)$.

- 1 First, it resolves the types τ_1 and τ_2 with the current solution ψ into τ'_1 and τ'_2 using the **type resolving** function `resolve`.
- 2 If one of τ'_1 or τ'_2 is a type variable, it checks recursive types using the **occurrence checking** and updates the solution of the type variable.
- 3 Otherwise, it checks τ'_1 and τ'_2 are equal or recursively unifies them.

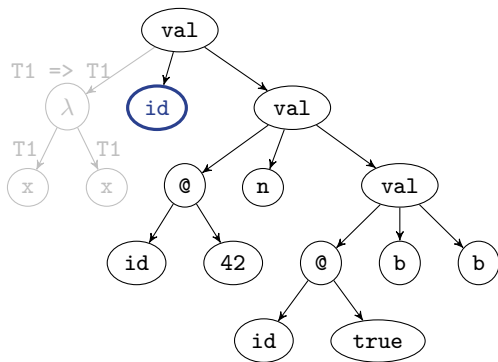
$$\text{unify}(\tau_1, \tau_2, \psi) = \begin{cases} \psi & \text{if } \tau_1' = \text{num} \wedge \tau_2' = \text{num} \\ \psi & \text{if } \tau_1' = \text{bool} \wedge \tau_2' = \text{bool} \\ \text{unify}(\tau_{1,r}, \tau_{2,r}, \text{unify}(\tau_{1,p}, \tau_{2,p}, \psi)) & \text{if } \tau_1' = (\tau_{1,p} \rightarrow \tau_{1,r}) \wedge \tau_2' = (\tau_{2,p} \rightarrow \tau_{2,r}) \\ \psi & \text{if } \tau_1' = \alpha = \tau_2' \\ \psi[\alpha \mapsto \tau_2'] & \text{if } \tau_1' = \alpha \wedge \neg \text{occur}(\alpha, \tau_2') \\ \psi[\alpha \mapsto \tau_1'] & \text{if } \tau_2' = \alpha \wedge \neg \text{occur}(\alpha, \tau_1') \end{cases}$$

where $\tau_1' = \text{resolve}(\tau_1, \psi)$ and $\tau_2' = \text{resolve}(\tau_2, \psi)$.

And, we can implement the **type unification** function in Scala as follows:

```
def unify(lty: Type, rty: Type, sol: Solution): Solution =
  (resolve(lty, sol), resolve(rty, sol)) match
  case (NumT, NumT) => sol
  case (BoolT, BoolT) => sol
  case (ArrowT(lpty, lrty), ArrowT(rpty, rrty)) =>
    unify(lrty, rrty, unify(lpty, rpty, sol))
  case (VarT(k), VarT(l)) if k == l => sol
  case (VarT(k), rty) if !occurs(k, rty, sol) => sol + (k -> Some(rty))
  case (lty, VarT(k)) if !occurs(k, lty, sol) => sol + (k -> Some(lty))
  case _ => error(s"Cannot unify ${lty.str} and ${rty.str}")
```

1. Type Checker and Typing Rules with Type Inference
 - Solutions for Type Constraints
 - Numbers
 - Additions
 - Conditionals
 - Immutable Variable Definitions and Identifier Lookup
 - Function Definitions
 - Recursive Function Definitions
 - Function Applications
2. Type Unification
 - Type Resolving
 - Occurrence Checking
 - Type Unification
3. Type Inference with Let-Polymorphism
 - Type Generalization
 - Type Instantiation



Type Environment

X	T
id	$[T1] \{ T1 \Rightarrow T1 \}$

Solution

X_α	T
T1	-

Let's **generalize** the type $T1 \Rightarrow T1$ into a **polymorphic type** for `id` with **type variable** $T1$ as a **type parameter**.

We call this **let-polymorphism** because it only introduces polymorphism for the let-binding (e.g., `val`).

We need to extend the **type environment** with **type schemes**, restricted forms of polymorphic types.

Type Environments $\Gamma \in \Gamma = \mathbb{X} \xrightarrow{\text{fin}} \mathbb{T}^\forall$

Type Schemes $\forall(\alpha^*). \tau = \tau^\forall \in \mathbb{T}^\forall = \mathbb{X}_\alpha^* \times \mathbb{T}$

Types $\mathbb{T} \ni \tau ::= \text{num} \mid \text{bool} \mid \tau \rightarrow \tau \mid \alpha$

Note that polymorphic types are not types in TIFAE, and **type schemes** are restricted forms of polymorphic types used in type environments.

We can define the **type environment** and **type schemes** in Scala:

```
// type environments
type TypeEnv = Map[String, TypeScheme]
// type schemes
case class TypeScheme(ks: List[Int], ty: Type)
```

```
def typeCheck(e: Expr, tenv: TypeEnv, sol: Solution): (Type, Solution) = e match
  ...

case Val(x, e, b) =>
  val (ety, sol1) = typeCheck(e, tenv, sol)
  val polyty = gen(ety, tenv, sol1)
  typeCheck(b, tenv + (x -> polyty), sol1)
```

$$\Gamma, \psi \vdash e : \tau, \psi$$

$$\tau\text{-Val} \frac{\Gamma, \psi_0 \vdash e_1 : \tau_1, \psi_1 \quad \text{gen}(\tau_1, \Gamma, \psi_1) = \tau_1^\forall \quad \Gamma[x : \tau_1^\forall], \psi_1 \vdash e_2 : \tau_2, \psi_2}{\Gamma, \psi_0 \vdash \text{val } x = e_1; e_2 : \tau_2, \psi_2}$$

We need to **generalize** the type τ_1 of the expression e_1 into a **type scheme** τ_1^\forall using the **type generalization** function `gen`. For example,

$$\text{gen}(\alpha \rightarrow \alpha, \emptyset, [\alpha \mapsto \bullet]) = \forall \alpha. (\alpha \rightarrow \alpha)$$

We can define the **type generalization** function gen as follows:

$$\boxed{\text{gen} : (\mathbb{T} \times \Gamma \times \Psi) \rightarrow \mathbb{T}^\forall}$$

$$\text{gen}(\tau, \Gamma, \psi) = \forall(\alpha_1, \dots, \alpha_m). \tau \quad \text{where} \quad \text{free}_\tau(\tau, \psi) \setminus \text{free}_\Gamma(\Gamma, \psi) = \{\alpha_1, \dots, \alpha_m\}$$

with the following definitions of **free type variables** in each component:

$$\boxed{\text{free}_\tau : (\mathbb{T} \times \Psi) \rightarrow \mathcal{P}(\mathbb{X}_\alpha)}$$

$$\text{free}_\tau(\tau, \psi) = \begin{cases} \{\alpha\} & \text{if } \tau = \alpha \wedge \psi(\alpha) = \bullet \\ \text{free}_\tau(\tau', \psi) & \text{if } \tau = \alpha \wedge \psi(\alpha) = \tau' \\ \text{free}_\tau(\tau_p, \psi) \cup \text{free}_\tau(\tau_r, \psi) & \text{if } \tau = (\tau_p \rightarrow \tau_r) \\ \emptyset & \text{otherwise} \end{cases}$$

$$\boxed{\text{free}_{\tau^\forall} : (\mathbb{T}^\forall \times \Psi) \rightarrow \mathcal{P}(\mathbb{X}_\alpha)}$$

$$\text{free}_{\tau^\forall}(\forall(\alpha_1, \dots, \alpha_m). \tau, \psi) = \text{free}_\tau(\tau, \psi) \setminus \{\alpha_1, \dots, \alpha_m\}$$

$$\boxed{\text{free}_\Gamma : (\Gamma \times \Psi) \rightarrow \mathcal{P}(\mathbb{X}_\alpha)}$$

$$\text{free}_\Gamma([x_1 : \tau_1^\forall, \dots, x_n : \tau_n^\forall], \psi) = \text{free}_{\tau_1^\forall}(\tau_1^\forall, \psi) \cup \dots \cup \text{free}_{\tau_n^\forall}(\tau_n^\forall, \psi)$$

We can define the **type generalization** function `gen` as follows:

$$\text{gen} : (\mathbb{T} \times \Gamma \times \Psi) \rightarrow \mathbb{T}^\forall$$

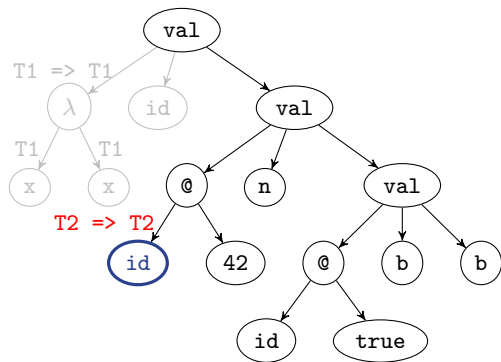
$$\text{gen}(\tau, \Gamma, \psi) = \forall(\alpha_1, \dots, \alpha_m). \tau \quad \text{where} \quad \text{free}_\tau(\tau, \psi) \setminus \text{free}_\Gamma(\Gamma, \psi) = \{\alpha_1, \dots, \alpha_m\}$$

Why do we need to subtract the free type variables $\text{free}_\Gamma(\Gamma, \psi)$ in the type environment Γ when generalizing the type τ ?

Consider the following example:

```
/* TIFAE */
x => {
  // tyenv = [x: T1] and solution = [T1 -> _] (T1 is free in tyenv)
  val z = x;           // z: T1 (0) not z: [T1] { T1 } (X)
  z                   // z: T1 (0) not z: T2 (X)
}
```

If we generalize the type $T1$ to $[T1] \{ T1 \Rightarrow T1 \}$ for z , the types of x and z will be different even though they have exactly the same value.



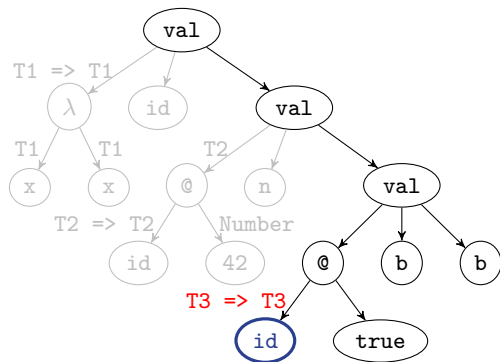
Type Environment

X	T
id	[T1] { T1 => T1 }

Solution

X_α	T
T1	-
T2	-

Let's define a new **type variable T2** to **instantiate** the **type variable T1**.
And, **substitute T1** with **T2**.



Type Environment

X	T
id	$[T1] \{ T1 \Rightarrow T1 \}$
n	$T2$

Solution

X_α	T
$T1$	-
$T2$	Number
$T3$	-

Let's define a new **type variable** $T3$ to **instantiate** the **type variable** $T1$.
 And, **substitute** $T1$ with $T3$.

```
def typeCheck(e: Expr, tenv: TypeEnv, sol: Solution): (Type, Solution) = e match
  ...

case Id(x) =>
  val ty = tenv.getOrElse(x, error(s"free identifier: $x"))
  inst(ty, sol)
```

$$\Gamma, \psi \vdash e : \tau, \psi$$

$$\tau\text{-Id} \frac{\Gamma(x) = \tau^{\forall} \quad \text{inst}(\tau^{\forall}, \psi) = (\tau, \psi')}{\Gamma, \psi \vdash x : \tau, \psi'}$$

We need to **instantiate** the type scheme τ^{\forall} with new type variables using the **type instantiation** function `inst`. For example,

$$\text{inst}(\forall \alpha. (\alpha \rightarrow \alpha), \emptyset) = (\beta \rightarrow \beta, [\beta \mapsto \bullet])$$

We can define the **type instantiation** function `inst` as follows:

$$\text{inst} : (\mathbb{T}^\forall \times \Psi) \rightarrow (\mathbb{T} \times \Psi)$$

$$\text{inst}(\forall(\alpha_1, \dots, \alpha_m). \tau, \psi) = (\text{subst}(\tau, \psi[\alpha_1 \mapsto \alpha'_1, \dots, \alpha_m \mapsto \alpha'_m]), \psi[\alpha'_1 \mapsto \bullet, \dots, \alpha'_m \mapsto \bullet])$$

where $\alpha'_1, \dots, \alpha'_m \notin \psi \wedge \forall 1 \leq i < j \leq m. \alpha'_i \neq \alpha'_j$

with the following **type substitution** function `subst`:

$$\text{subst} : (\mathbb{T} \times \Psi) \rightarrow \mathbb{T}$$

$$\text{subst}(\tau, \psi) = \begin{cases} \text{subst}(\tau', \psi) & \text{if } \tau = \alpha \wedge \psi(\alpha) = \tau' \\ \text{subst}(\tau_p, \psi) \rightarrow \text{subst}(\tau_r, \psi) & \text{if } \tau = (\tau_p \rightarrow \tau_r) \\ \tau & \text{otherwise} \end{cases}$$

1. Type Checker and Typing Rules with Type Inference

- Solutions for Type Constraints

- Numbers

- Additions

- Conditionals

- Immutable Variable Definitions and Identifier Lookup

- Function Definitions

- Recursive Function Definitions

- Function Applications

2. Type Unification

- Type Resolving

- Occurrence Checking

- Type Unification

3. Type Inference with Let-Polymorphism

- Type Generalization

- Type Instantiation

<https://github.com/ku-plrg-classroom/docs/tree/main/cose212/tifae>

- Please see above document on GitHub:
 - Implement `typeCheck` function.
 - Implement `interp` function.
- It is just an exercise, and you **don't need to submit** anything.
- However, some exam questions might be related to this exercise.

- **Date:** 18:30 – 21:00 (150 min.), December 18 (Wed.).
- **Location:** 205, Woojung Hall of Informatics (우정정보관)
- **Coverage:** Lectures 14 – 26
- **Format:** closed book and closed notes
 - Fill-in-the-blank questions about the PL concepts.
 - Write the evaluation results of given expressions.
 - Draw derivation trees of given expressions.
 - Define the syntax or semantics of extended language features.
 - Define typing rules for the given language features.
 - etc.
- Note that there is **no class** on **December 16 (Mon.)**.

- Course Review

Jihyeok Park
jihyeok_park@korea.ac.kr
<https://plrg.korea.ac.kr>