Lecture 26 – Type Inference (2)
COSE212: Programming Languages

Jihyeok Park

A PLRG

2023 Fall

- **Type inference** is the process of automatically inferring the types of expressions.

- **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
```

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

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

- **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
```

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

```
/* RFAE */ 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.

OPLRG

- **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
```

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

```
/* RFAE */ 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

Contents

Contents

◆PLRG

Type Checker and Typing Rules

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:

```scala
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.

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

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

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
```
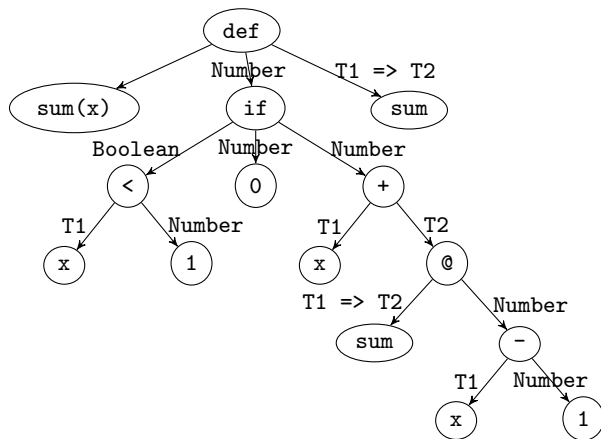

Type Environment

$\mathbb{X}$	$\mathbb{T}$
x	T1
sum	T1 => T2

Solution

$\mathbb{X}_\alpha$	$\mathbb{T}$
T1	Number
T2	Number

Solutions for Type Constraints

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

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

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

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

Note that $\bot$ (None) represents a **not yet solved** (**free**) type variable.

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

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

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

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

Note that $\bot$ (`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) = ???
```

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

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)
    (ty, sol4)
```

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

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

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

Conditionals

```scala
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)
```

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

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

Immutable Variable Defs. and Identifier Lookup

```
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) => tenv.getOrElse(x, error(s"free identifier: $x"))
```

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

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

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

Function Definitions

```
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)
```

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

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

Recursive Function Definitions

```scala
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)
```

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

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

```
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)
```

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

$$\tau-\text{App} \ \frac{\Gamma, \psi \vdash e_f : \tau_f, \psi_f \qquad \Gamma, \psi_f \vdash e_a : \tau_a, \psi_a}{\alpha_r \notin \psi_a \qquad \texttt{unify}(\tau_a \to \alpha_r, \tau_f, \psi_a[\alpha_r \mapsto \bot]) = \psi'}{\Gamma, \psi \vdash e_1(e_2) : \alpha_r, \psi'}$$

Contents

Type Unification

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.

$$\texttt{unify} : (\mathbb{T} \times \mathbb{T} \times \Psi) \rightharpoonup \Psi$$

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

$$\texttt{unify}(\alpha, \texttt{num}, \varnothing) = [\alpha \mapsto \texttt{num}]$$

Type Unification

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.

$$\texttt{unify} : (\mathbb{T} \times \mathbb{T} \times \Psi) \rightharpoonup \Psi$$

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

$$\texttt{unify}(\alpha, \texttt{num}, \varnothing) = [\alpha \mapsto \texttt{num}]$$

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 **cyclic types**.

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

$$\texttt{unify}(\alpha_1, \alpha_2, \psi_1) = \psi_2 \qquad\qquad \texttt{unify}(\alpha_1, \texttt{num}, \psi_2) = \psi_3$$

$$\psi_1 = \begin{array}{|c|c|}
\hline
\multicolumn{2}{|c|}{\text{Solution}} \\
\hline
\mathbb{X}_\alpha & \mathbb{T} \\
\hline
\alpha_1 & \bot \\
\hline
\alpha_2 & \bot \\
\hline
\end{array}
\qquad
\psi_2 = \begin{array}{|c|c|}
\hline
\multicolumn{2}{|c|}{\text{Solution}} \\
\hline
\mathbb{X}_\alpha & \mathbb{T} \\
\hline
\alpha_1 & \alpha_2 \\
\hline
\alpha_2 & \bot \\
\hline
\end{array}
\qquad
\psi_3 = \begin{array}{|c|c|}
\hline
\multicolumn{2}{|c|}{\text{Solution}} \\
\hline
\mathbb{X}_\alpha & \mathbb{T} \\
\hline
\alpha_1 & \texttt{num} \\
\hline
\alpha_2 & \bot \\
\hline
\end{array}$$

Type Resolving

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

$$\text{unify}(\alpha_1, \alpha_2, \psi_1) = \psi_2 \qquad\qquad \text{unify}(\alpha_1, \text{num}, \psi_2) = \psi_3$$

$$\psi_1 = \begin{array}{c}\text{Solution}\\ \begin{array}{|c|c|}\hline \mathbb{X}_\alpha & \mathbb{T} \\ \hline\hline \alpha_1 & \bot \\ \hline \alpha_2 & \bot \\ \hline\end{array}\end{array} \qquad \psi_2 = \begin{array}{c}\text{Solution}\\ \begin{array}{|c|c|}\hline \mathbb{X}_\alpha & \mathbb{T} \\ \hline\hline \alpha_1 & \alpha_2 \\ \hline \alpha_2 & \bot \\ \hline\end{array}\end{array} \qquad \psi_3 = \begin{array}{c}\text{Solution}\\ \begin{array}{|c|c|}\hline \mathbb{X}_\alpha & \mathbb{T} \\ \hline\hline \alpha_1 & \text{num} \\ \hline \alpha_2 & \bot \\ \hline\end{array}\end{array}$$

Unfortunately, we cannot know that α_2 are num with the solution ψ_3.

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

$$\texttt{unify}(\alpha_1, \alpha_2, \psi_1) = \psi_2 \qquad \texttt{unify}(\alpha_1, \texttt{num}, \psi_2) = \psi_3$$

$$\psi_1 = \begin{array}{|c|c|} \hline \multicolumn{2}{|c|}{\text{Solution}} \\ \hline \mathbb{X}_\alpha & \mathbb{T} \\ \hline \alpha_1 & \bot \\ \hline \alpha_2 & \bot \\ \hline \end{array} \qquad \psi_2 = \begin{array}{|c|c|} \hline \multicolumn{2}{|c|}{\text{Solution}} \\ \hline \mathbb{X}_\alpha & \mathbb{T} \\ \hline \alpha_1 & \alpha_2 \\ \hline \alpha_2 & \bot \\ \hline \end{array} \qquad \psi_3 = \begin{array}{|c|c|} \hline \multicolumn{2}{|c|}{\text{Solution}} \\ \hline \mathbb{X}_\alpha & \mathbb{T} \\ \hline \alpha_1 & \texttt{num} \\ \hline \alpha_2 & \bot \\ \hline \end{array}$$

Unfortunately, we cannot know that α_2 are num with the solution ψ_3.

We need to **resolve** the type variable α_1 to find its **representative type** and update its solution to num to deal with the **type aliasing**.

$$\texttt{unify}(\texttt{resolve}(\alpha_1, \psi_2), \texttt{num}, \psi_2) = \psi_3' = \begin{array}{|c|c|} \hline \multicolumn{2}{|c|}{\text{Solution}} \\ \hline \mathbb{X}_\alpha & \mathbb{T} \\ \hline \alpha_1 & \alpha_2 \\ \hline \alpha_2 & \texttt{num} \\ \hline \end{array}$$

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

$$\texttt{resolve} : (\mathbb{T} \times \Psi) \to \mathbb{T}$$

$$\texttt{resolve}(\tau, \psi) = \left\{ \begin{array}{ll} \texttt{resolve}(\tau', \psi) & \text{if } \tau = \alpha \wedge \psi(\alpha) = \tau' \\ \tau & \text{otherwise} \end{array} \right.$$

and implement it in Scala as follows:

```scala
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
```

OPLRG

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

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

It actually fails because the type variable α_1 occurs in the type $\text{num} \to \alpha_1$, which means it requires **cyclic types** not supported in our type system.

Occurrence Checking

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

$$\mathtt{unify}(\alpha_1, \mathtt{num} \to \alpha_1, \psi) = \psi'$$

It actually fails because the type variable α_1 occurs in the type $\mathtt{num} \to \alpha_1$, which means it requires **cyclic types** not supported in our type system.

Let's define the **occurrence checking** function as follows:

$$\boxed{\mathtt{occur} : (\mathbb{X}_\alpha \times \mathbb{T} \times \Psi) \to \mathtt{bool}}$$

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

and implement it in Scala as follows:

```scala
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:

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

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

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 only one of them (τ_1' or τ_2') is a type variable, it checks cyclic types using the **occurrence checking** function occur.

3 Then, it unifies types τ_1' and τ_2' and updates the solution ψ.
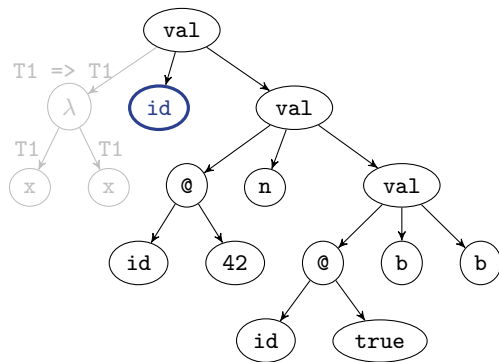
Type Unification

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

```scala
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}")
```

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

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

Contents

Type Environment

$\mathbb{X}$	$\mathbb{T}$
id	[T1] { T1 => T1 }

Solution

$\mathbb{X}_\alpha$	$\mathbb{T}$
T1	–

Let's **generalize** the type `T1 => 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`).

Type Generalization

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

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

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

Type Generalization

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

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

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

and the **free type variables** in each component as follows:

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

$$\text{free}_\tau(\tau, \psi) = \begin{cases} \text{free}_{\tau'}(\tau', \psi) & \text{if } \tau = \alpha \land \psi(\alpha) = \tau' \\ \{\alpha\} & \text{if } \tau = \alpha \land \psi(\alpha) = \bot \\ \text{free}_{\tau_p}(\tau_p, \psi) \cup \text{free}_{\tau_r}(\tau_r, \psi) & \text{if } \tau = (\tau_p \to \tau_r) \\ \varnothing & \text{otherwise} \end{cases}$$

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

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

$$\boxed{\text{free}_\Gamma : (\mathbb{\Gamma} \times \Psi) \to \mathcal{P}(\mathbb{X}_\alpha)}$$
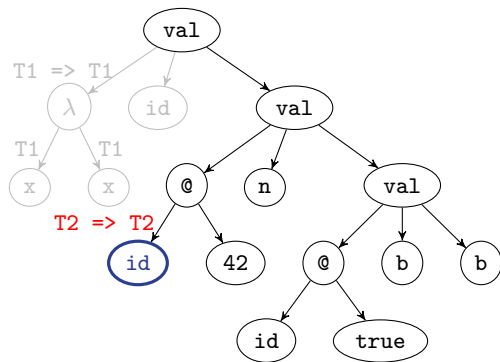
$$\text{free}_\Gamma([x_1 : \tau_1^\forall, \dots, x_n : \tau_n^\forall], \psi) = \text{free}_{\tau_1^\forall}(\tau_1^\forall, \psi) \cup \dots \cup \text{free}_{\tau_n^\forall}(\tau_n^\forall, \psi)$$

Immutable Variable Defs. with Type Generalization ◭PLRG

```
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)
```

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

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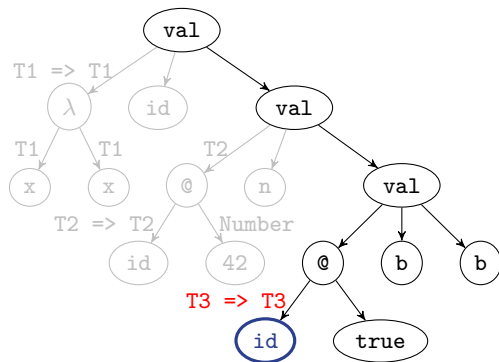
Type Environment

$\mathbb{X}$	$\mathbb{T}$
id	[T1] { T1 => T1 }

Solution

$\mathbb{X}_\alpha$	$\mathbb{T}$
T1	–
T2	–

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

Type Environment

$\mathbb{X}$	$\mathbb{T}$
`id`	`[T1] { T1 => T1 }`
`n`	`T2`

Solution

$\mathbb{X}_\alpha$	$\mathbb{T}$
`T1`	–
`T2`	`Number`
`T3`	–

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

Type Instantiation

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

$$\texttt{inst} : (\mathbb{T}^\forall \times \Psi) \to (\mathbb{T} \times \Psi)$$

$$
\begin{aligned}
\texttt{inst}(\forall \alpha_1, \ldots, &\alpha_m.\tau, \psi) = (\\
&\texttt{subst}(\tau, \psi[\alpha_1 \mapsto \alpha_1', \ldots, \alpha_m \mapsto \alpha_m']),\\
&\psi[\alpha_1' \mapsto \bot, \ldots, \alpha_m' \mapsto \bot]\\
)&
\end{aligned}
$$

where $\quad \alpha_1', \ldots, \alpha_m' \notin \psi \land \forall 1 \le i < j \le m.\ \alpha_i' \ne \alpha_j'$

Type Instantiation

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

$$\boxed{\texttt{inst} : (\mathbb{T}^\forall \times \Psi) \to (\mathbb{T} \times \Psi)}$$

$$
\begin{aligned}
\texttt{inst}(\forall \alpha_1, \ldots, \alpha_m.\tau, \psi) = (\ &\\
\texttt{subst}(\tau, \psi[\alpha_1 \mapsto \alpha_1', \ldots, \alpha_m \mapsto \alpha_m']),&\\
\psi[\alpha_1' \mapsto \bot, \ldots, \alpha_m' \mapsto \bot]&\\
)&
\end{aligned}
$$

where $\quad \alpha_1', \ldots, \alpha_m' \notin \psi \wedge \forall 1 \leq i < j \leq m. \; \alpha_i' \neq \alpha_j'$

and the **type substitution** function subst as follows:

$$\boxed{\texttt{subst} : (\mathbb{T} \times \Psi) \to \mathbb{T}}$$

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

```
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)
```

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

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

Summary

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

Exercise #15

- Please see this document[1] 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.

---

[1]https://github.com/ku-plrg-classroom/docs/tree/main/cose212/tifae.

- The final exam will be given in class.
- **Date:** 13:30-14:45 (1 hour 15 minutes), December 20 (Wed.).
- **Location:** 535, Asan Science Building (아산이학관)
- **Coverage:** Lectures 14 – 26
- **Format:** 7–9 questions with closed book and closed notes
    - Fill in the blank in a Scala code snippet.
    - Define the syntax or semantics of extended language features.
    - Write the evaluation results of given expressions.
    - Yes/No questions about concepts in programming languages.
    - etc.
- Note that there is **no class** on **December 18 (Mon.)**.

Next Lecture

- Course Review

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