Lecture 24 – Subtype Polymorphism

COSE212: Programming Languages

Jihyeok Park

PLRG

2024 Fall

Recall

- **Polymorphism** is to use a single entity as **multiple types**, and there are various kinds of polymorphism:
  - **Parametric polymorphism**
  - **Subtype polymorphism**
  - **Ad-hoc polymorphism**
  - ...

Recall

- **Polymorphism** is to use a single entity as **multiple types**, and there are various kinds of polymorphism:
    - **Parametric polymorphism**
    - **Subtype polymorphism**
    - **Ad-hoc polymorphism**
    - ...
- **Parametric polymorphism** is a form of polymorphism by introducing **type variables** and instantiating them with **type arguments**.
- **PTFAE** – TFAE with **parametric polymorphism**.

Recall

- **Polymorphism** is to use a single entity as **multiple types**, and there are various kinds of polymorphism:
    - **Parametric polymorphism**
    - **Subtype polymorphism**
    - **Ad-hoc polymorphism**
    - ...
- **Parametric polymorphism** is a form of polymorphism by introducing **type variables** and instantiating them with **type arguments**.
- **PTFAE** – TFAE with **parametric polymorphism**.
- In this lecture, we will learn **subtype polymorphism**.

Recall

- **Polymorphism** is to use a single entity as **multiple types**, and there are various kinds of polymorphism:
  - **Parametric polymorphism**
  - **Subtype polymorphism**
  - **Ad-hoc polymorphism**
  - ...
- **Parametric polymorphism** is a form of polymorphism by introducing **type variables** and instantiating them with **type arguments**.
- **PTFAE** – TFAE with **parametric polymorphism**.
- In this lecture, we will learn **subtype polymorphism**.
- **STFAE** – TFAE with **subtype polymorphism**.
  - **Interpreter** and **Natural Semantics**
  - **Type Checker** and **Typing Rules**

Contents

Contents

Subtype Polymorphism

To easily explain **subtype polymorphism**, let's support **records** and **record types** to TFAE. (Also, optional type annotations for `val`.)

```
/* STFAE */
// A record with two fields `a` and `b` whose types are `Number`
val x: {a: Number, b: Number} = {a=1, b=2}
x.a    // Access the field `a` of `x` and evaluate to `1`
```

Subtype Polymorphism

To easily explain **subtype polymorphism**, let's support **records** and **record types** to TFAE. (Also, optional type annotations for `val`.)

```
/* STFAE */
// A record with two fields `a` and `b` whose types are `Number`
val x: {a: Number, b: Number} = {a=1, b=2}
x.a    // Access the field `a` of `x` and evaluate to `1`
```

Consider the following expression:

```
/* STFAE */
val f = (x: ???) => x.a
f({a=1}) + f({a=2, b=3}) + f({c=4, a=5})
```

Subtype Polymorphism

To easily explain **subtype polymorphism**, let's support **records** and **record types** to TFAE. (Also, optional type annotations for `val`.)

```
/* STFAE */
// A record with two fields `a` and `b` whose types are `Number`
val x: {a: Number, b: Number} = {a=1, b=2}
x.a    // Access the field `a` of `x` and evaluate to `1`
```

Consider the following expression:

```
/* STFAE */
val f = (x: ???) => x.a
f({a=1}) + f({a=2, b=3}) + f({c=4, a=5})
```

Unfortunately, we cannot assign any type to `x` because the type of `x` should be ① `{a: Number}`, ② `{a: Number, b: Number}`, and ③ `{c: Number, a: Number}`, simultaneously.

Subtype Polymorphism

To easily explain **subtype polymorphism**, let's support **records** and **record types** to TFAE. (Also, optional type annotations for `val`.)

```
/* STFAE */
// A record with two fields `a` and `b` whose types are `Number`
val x: {a: Number, b: Number} = {a=1, b=2}
x.a    // Access the field `a` of `x` and evaluate to `1`
```

Consider the following expression:

```
/* STFAE */
val f = (x: ???) => x.a
f({a=1}) + f({a=2, b=3}) + f({c=4, a=5})
```

Unfortunately, we cannot assign any type to `x` because the type of `x` should be ① `{a: Number}`, ② `{a: Number, b: Number}`, and ③ `{c: Number, a: Number}`, simultaneously.

How can we resolve this problem?

Subtype Polymorphism

To easily explain **subtype polymorphism**, let's support **records** and **record types** to TFAE. (Also, optional type annotations for `val`.)

```
/* STFAE */
// A record with two fields `a` and `b` whose types are `Number`
val x: {a: Number, b: Number} = {a=1, b=2}
x.a    // Access the field `a` of `x` and evaluate to `1`
```

Consider the following expression:

```
/* STFAE */
val f = (x: ???) => x.a
f({a=1}) + f({a=2, b=3}) + f({c=4, a=5})
```

Unfortunately, we cannot assign any type to `x` because the type of `x` should be 1 `{a: Number}`, 2 `{a: Number, b: Number}`, and 3 `{c: Number, a: Number}`, simultaneously.

How can we resolve this problem? **Subtype Polymorphism**!

Subtype Polymorphism

Definition (Subtype Polymorphism)

Subtype polymorphism is a form of polymorphism by introducing **subtype relations** between types.

Subtype Polymorphism

Definition (Subtype Polymorphism)

Subtype polymorphism is a form of polymorphism by introducing **subtype relations** between types.

```
/* STFAE */
val f = (x: {a: Number}) => x.a     // Allow subtypes of `{a: Number}`
f({a=1}) + f({a=2, b=3}) + f({c=4, a=5})
```

All the following types are **subtypes** of `{a: Number}`:

`{a: Number}` `{a: Number, b: Number}`
`{c: Number, a: Number}` ...

Subtype Polymorphism

Definition (Subtype Polymorphism)

Subtype polymorphism is a form of polymorphism by introducing **subtype relations** between types.

```
/* STFAE */
val f = (x: {a: Number}) => x.a     // Allow subtypes of `{a: Number}`
f({a=1}) + f({a=2, b=3}) + f({c=4, a=5})
```

All the following types are **subtypes** of `{a: Number}`:

`{a: Number}` `{a: Number, b: Number}`
`{c: Number, a: Number}` ...

It corresponds to the **subset relation** between sets in mathematics, and most programming languages support **subtype polymorphism**.

Subtype Polymorphism

Definition (Subtype Polymorphism)

Subtype polymorphism is a form of polymorphism by introducing **subtype relations** between types.

```
/* STFAE */
val f = (x: {a: Number}) => x.a      // Allow subtypes of `{a: Number}`
f({a=1}) + f({a=2, b=3}) + f({c=4, a=5})
```

All the following types are **subtypes** of `{a: Number}`:

`{a: Number}` `{a: Number, b: Number}`
`{c: Number, a: Number}` ...

It corresponds to the **subset relation** between sets in mathematics, and most programming languages support **subtype polymorphism**.

Subtype relations could be defined for other types (e.g., functions, lists, pairs, data types etc.) as well.

Contents

STFAE – TFAE with Subtype Polymorphism

Now, let's extend TFAE into STFAE to support **subtype polymorphism** with **records** and optional type annotations for `val`.

```
/* STFAE */
val x: {a: Number} = {a=1}                  // A record with a field `a`
x.a                                         // Access the field `a` of `x`
val x: {a: Top, b: Top} = {a=1, b=x}        // `Top` is the top type
val x: Bot = exit                           // Exit the program
...                                         // Not evaluated after `exit`
```

STFAE – TFAE with Subtype Polymorphism

Now, let's extend TFAE into STFAE to support **subtype polymorphism** with **records** and optional type annotations for `val`.

```
/* STFAE */
val x: {a: Number} = {a=1}              // A record with a field `a`
x.a                                     // Access the field `a` of `x`
val x: {a: Top, b: Top} = {a=1, b=x}   // `Top` is the top type
val x: Bot = exit                       // Exit the program
...                                     // Not evaluated after `exit`
```

For STFAE, we need to extend **expressions** of TFAE with

1. **Optional Type Annotations for `val`**
2. **Records**
3. **Field Accesses**
4. **Exit** (to immediately exit the program)
5. **Record Types**
6. **Bottom Type** (corresponding to the empty set)
7. **Top Type** (corresponding to the universal set)

Concrete Syntax

1. **Optional Type Annotations for** `val`
2. **Records**
3. **Field Accesses**
4. **Exit** (to immediately exit the program)
5. **Record Types**
6. **Bottom Type** (corresponding to the empty set)
7. **Top Type** (corresponding to the universal set)

Concrete Syntax

1. **Optional Type Annotations for** `val`
2. **Records**
3. **Field Accesses**
4. **Exit** (to immediately exit the program)
5. **Record Types**
6. **Bottom Type** (corresponding to the empty set)
7. **Top Type** (corresponding to the universal set)

We can extend the **concrete syntax** of TFAE as follows:

```
<expr>      ::= ...
              | "val" <id> [ ":" <type> ]? "=" <expr> ";"? <expr>
              | "{" [<id> "=" <expr>]*{","} "}"
              | <expr> "." <id>
              | "exit"
<type>      ::= ...
              | "{" [<id> ":" <type>]*{","} "}"
              | "Bot"
              | "Top"
```

Abstract Syntax

PLRG

$$
\begin{array}{llll}
\text{Expressions} & \mathbb{E} \ni e ::= \ldots & \mid \{[x = e]^*\} & (\texttt{Record}) \\
& \mid \texttt{val}\ x\ [: \tau]^? = e;\ e\ (\texttt{Val}) & \mid e.x & (\texttt{Access}) \\
& & \mid \texttt{exit} & (\texttt{Exit})
\end{array}
$$

$$
\begin{array}{lll}
\text{Types} & \mathbb{T} \ni \tau ::= \ldots & \mid \bot\ (\texttt{BotT}) \\
& \mid \{[x : \tau]^*\}\ (\texttt{RecordT}) & \mid \top\ (\texttt{TopT})
\end{array}
$$

```
enum Expr:
  ...
  case Val(name: String, tyOpt: Option[Type], init: Expr, body: Expr)
  case Record(fields: List[(String, Expr)])
  case Access(record: Expr, field: String)
  case Exit

enum Type:
  ...
  case RecordT(fields: Map[String, Type])
  case BotT
  case TopT
```

Contents

Interpreter and Natural Semantics

For STFAE, we need to 1) implement the **interpreter** with environments:

```
def interp(expr: Expr, env: Env): Value = ???
```

and 2) define the **natural semantics** with environments:

$$\boxed{\sigma \vdash e \Rightarrow v}$$

Interpreter and Natural Semantics

For STFAE, we need to 1) implement the **interpreter** with environments:

```
def interp(expr: Expr, env: Env): Value = ???
```

and 2) define the **natural semantics** with environments:

$$\boxed{\sigma \vdash e \Rightarrow v}$$

with a new kind of values called **record values**:

$$
\begin{array}{lrll}
\text{Values} & \mathbb{V} \ni v ::= & n & (\texttt{NumV}) \\
& | & \langle \lambda x.e, \sigma \rangle & (\texttt{CloV}) \\
& | & \{[x = v]^*\} & (\texttt{RecordV})
\end{array}
$$

```
enum Value:
  case NumV(number: BigInt)
  case CloV(param: String, body: Expr, env: Env)
  case RecordV(fields: Map[String, Value])
```

```scala
def interp(expr: Expr, env: Env): Value = expr match
  ...

  case Record(fs) =>
    RecordV(fs.map { case (f, e) => (f, interp(e, env)) }.toMap)

  case Access(r, f) => interp(r, env) match
    case RecordV(fs) => fs.getOrElse(f, error(s"no such field: $f"))
    case v           => error(s"not a record: ${v.str}")
```

$$\boxed{\sigma \vdash e \Rightarrow v}$$

$$\texttt{Record}\ \frac{\sigma \vdash e_1 \Rightarrow v_1 \qquad \dots \qquad \sigma \vdash e_n \Rightarrow v_n}{\sigma \vdash \{x_1 = e_1, \dots, x_n = e_n\} \Rightarrow \{x_1 = v_1, \dots, x_n = v_n\}}$$

$$\texttt{Access}\ \frac{\sigma \vdash e \Rightarrow \{\dots, x = v, \dots\}}{\sigma \vdash e.x \Rightarrow v}$$

Exit

```
def interp(expr: Expr, env: Env): Value = expr match
  ...
  case Exit => error("exit")
```

$$\boxed{\sigma \vdash e \Rightarrow v}$$

There is no rule for `exit` because it cannot produce any value.

Exit

```
def interp(expr: Expr, env: Env): Value = expr match
  ...
  case Exit => error("exit")
```

$$\boxed{\sigma \vdash e \Rightarrow v}$$

There is no rule for `exit` because it cannot produce any value.

We cannot draw the derivation tree for the following expression:

```
/* STFAE */ 1 + exit
```

Exit

PLRG

```scala
def interp(expr: Expr, env: Env): Value = expr match
  ...
  case Exit => error("exit")
```

$$\boxed{\sigma \vdash e \Rightarrow v}$$

There is no rule for `exit` because it cannot produce any value.

We cannot draw the derivation tree for the following expression:

```
/* STFAE */ 1 + exit
```

However, we can draw the derivation tree for the following expression:

```
/* STFAE */ (x: Number) => 1 + exit
```

$$\texttt{Fun}\ \frac{}{\varnothing \vdash \lambda x{:}\texttt{num}.(1 \texttt{ + } \texttt{exit}) \Rightarrow \langle \lambda x.(1 \texttt{ + } \texttt{exit}), \varnothing \rangle}$$

Immutable Variable Definition

```
def interp(expr: Expr, env: Env): Value = expr match
  ...

  case Val(name, _, expr, body) =>
    interp(body, env + (name -> interp(expr, env)))
```

$$\boxed{\sigma \vdash e \Rightarrow v}$$

$$\texttt{Val}\ \frac{\sigma \vdash e_1 \Rightarrow v_1 \qquad \sigma[x \mapsto v_1] \vdash e_2 \Rightarrow v_2}{\sigma \vdash \texttt{val}\ x\ \texttt{=}\ e_1;\ e_2 \Rightarrow v_2}$$

$$\texttt{Val}_\tau\ \frac{\sigma \vdash e_1 \Rightarrow v_1 \qquad \sigma[x \mapsto v_1] \vdash e_2 \Rightarrow v_2}{\sigma \vdash \texttt{val}\ x : \tau_0\ \texttt{=}\ e_1;\ e_2 \Rightarrow v_2}$$

Contents

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

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

and ❷ 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.

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

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

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

Records

```
def typeCheck(expr: Expr, tenv: TypeEnv): Type = expr match
  ...
  case Record(fields) =>
    RecordT(fields.map { case (f, e) => (f, typeCheck(e, tenv)) }.toMap)
```

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

$$\tau\text{–Record}\ \frac{\Gamma \vdash e_1 : \tau_1 \qquad \ldots \qquad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash \{x_1 = e_1, \ldots, x_n = e_n\} : \{x_1 : \tau_1, \ldots, x_n : \tau_n\}}$$

Field Accesses

```
def typeCheck(expr: Expr, tenv: TypeEnv): Type = expr match
  ...
  case Access(record, f) => typeCheck(record, tenv) match
    case RecordT(fs) => fs.getOrElse(f, error(s"no such field: $f"))
    case ty          => error(s"not a record type: ${ty.str}")
```

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

$$\tau\text{-}\texttt{Access}\ \frac{\Gamma \vdash e : \{\ldots, x : \tau, \ldots\}}{\Gamma \vdash e.x : \tau}$$

Exit

```
def typeCheck(expr: Expr, tenv: TypeEnv): Type = expr match
  ...
  case Exit => BotT
```

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

$$\tau\text{–}\texttt{Exit}\ \frac{}{\Gamma \vdash \texttt{exit} : \bot}$$

Immutable Variable Definition

```
def typeCheck(expr: Expr, tenv: TypeEnv): Type = expr match
  ...
  case Val(name, tyOpt, expr, body) =>
    val ty = typeCheck(expr, tenv)


    typeCheck(body, tenv + (name -> ty))
```

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

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

Immutable Variable Definition

```scala
def typeCheck(expr: Expr, tenv: TypeEnv): Type = expr match
  ...
  case Val(name, tyOpt, expr, body) =>
    val ty = typeCheck(expr, tenv)
    tyOpt.map(givenTy => mustEqual(ty, givenTy))
    val nameTy = tyOpt.getOrElse(ty)
    typeCheck(body, tenv + (name -> nameTy))
```

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

$$\tau\text{–}\mathtt{Val}\ \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma[x : \tau_1] \vdash e_2 : \tau_2}{\Gamma \vdash \mathtt{val}\ x\ \mathtt{=}\ e_1;\ e_2 : \tau_2}$$

$$\tau\text{–}\mathtt{Val}_\tau\ \frac{\Gamma \vdash e_1 : \tau_1 \qquad \tau_1 = \tau_0 \qquad \Gamma[x : \tau_0] \vdash e_2 : \tau_2}{\Gamma \vdash \mathtt{val}\ x : \tau_0\ \mathtt{=}\ e_1;\ e_2 : \tau_2}$$

Immutable Variable Definition

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

$$\tau\text{–}\texttt{Val}_\tau \; \frac{\Gamma \vdash e_1 : \tau_1 \qquad \tau_1 = \tau_0 \qquad \Gamma[x : \tau_0] \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{val}\ x : \tau_0\ \texttt{=}\ e_1;\ e_2 : \tau_2}$$

Consider the following example:

```
/* STFAE */
val x: {a: Number} = {a=2, b=3}; x.a
```

It fails to type check because:

$$\texttt{\{a: Number, b: Number\}} \neq \texttt{\{a: Number\}}$$

Immutable Variable Definition

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

$$\tau\text{–}\mathtt{Val}_\tau \; \frac{\Gamma \vdash e_1 : \tau_1 \qquad \tau_1 = \tau_0 \qquad \Gamma[x : \tau_0] \vdash e_2 : \tau_2}{\Gamma \vdash \mathtt{val}\ x : \tau_0\ \mathtt{=}\ e_1;\ e_2 : \tau_2}$$

Consider the following example:

```
/* STFAE */
val x: {a: Number} = {a=2, b=3}; x.a
```

It fails to type check because:

$$\mathtt{\{a:\ Number,\ b:\ Number\}} \neq \mathtt{\{a:\ Number\}}$$

Let's apply **subtype polymorphism** to fix this problem by introducing a **subtype relation** ($<:$) between types.

Contents

Subtype Relation

To support **subtype polymorphism**, we need to define a **subtype relation** $<:$ between types.

$$\boxed{\tau <: \tau}$$

$\tau <: \tau'$ denotes τ is a **subtype** of τ' (or τ' is a **super type** of τ).

Subtype Relation

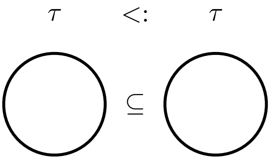
To support **subtype polymorphism**, we need to define a **subtype relation** $<:$ between types.

$$\boxed{\tau <: \tau}$$

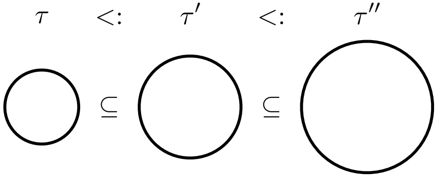
$\tau <: \tau'$ denotes τ is a **subtype** of τ' (or τ' is a **super type** of τ).

First, **subtype relation** is **reflexive** and **transitive**:

$$\frac{}{\tau <: \tau} \qquad \qquad \frac{\tau <: \tau' \qquad \tau' <: \tau''}{\tau <: \tau''}$$



Reflexivity



Transitivity

$$\boxed{\tau <: \tau}$$

The **bottom type** $\bot$ and the **top type** $\top$ represent the **empty set** of values and the **universal set** of values, respectively.

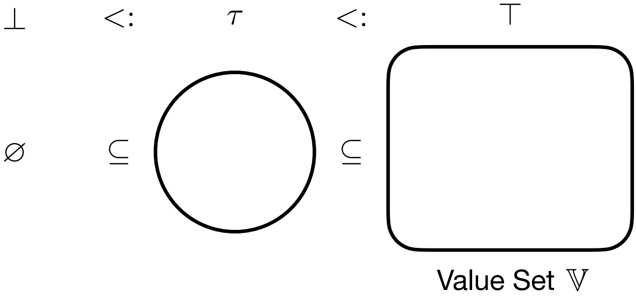
Subtype Relation – Bottom Type and Top Type

$$\boxed{\tau <: \tau}$$

The **bottom type** $\bot$ and the **top type** $\top$ represent the **empty set** of values and the **universal set** of values, respectively.

Thus, $\bot$ is a subtype of any type, and any type is a subtype of $\top$:

$$\frac{}{\bot <: \tau} \qquad \frac{}{\tau <: \top}$$



Subtype Relation – Record Types (1)

$$\boxed{\tau <: \tau}$$

Let's consider the subtype relation between **record types**.

```
/* STFAE */
val f = (x: {a: Number}) => x.a
val y: {a: Number, b: Number} = {a = 1, b = 2}
f(y)
```

If we **add** any new field to a record type, the resulting type should be a subtype of the original type.

$$\frac{}{\{x_1 : \tau_1, \ldots, x_n : \tau_n, x : \tau\} <: \{x_1 : \tau_1, \ldots, x_n : \tau_n\}}$$

Subtype Relation – Record Types (2)

$$\boxed{\tau <: \tau}$$

Let's consider the subtype relation between **record types**.

```
/* STFAE */
val f = (x: {a: Top, b: Top}) => x
val x: {a: Number, b: Number} = {a = 1, b = 2}
f(x)
```

If all fields of a record type are **subtypes** of the corresponding fields of another record type, the resulting type should be a subtype of the other.

$$\frac{\tau_1 <: \tau_1' \qquad \ldots \qquad \tau_n <: \tau_n'}{\{x_1 : \tau_1, \ldots, x_n : \tau_n\} <: \{x_1 : \tau_1', \ldots, x_n : \tau_n'\}}$$

Subtype Relation – Record Types (3)

$$\boxed{\tau <: \tau}$$

Let's consider the subtype relation between **record types**.

```
/* STFAE */
val f = (x: {a: Number, b: Number}) => x
val x: {b: Number, a: Number} = {a = 1, b = 2}
f(x)
```

If the fields of a record type is a **permutation** of the fields of another record type, the resulting type should be a subtype of the other.

$$\frac{\{x_1 : \tau_1, \ldots, x_n : \tau_n\} \text{ is a permutation of } \{x'_1 : \tau'_1, \ldots, x'_n : \tau'_n\}}{\{x_1 : \tau_1, \ldots, x_n : \tau_n\} <: \{x'_1 : \tau'_1, \ldots, x'_n : \tau'_n\}}$$

Subtype Relation – Function Types

$$\boxed{\tau <: \tau}$$

Let's consider the subtype relation between **function types**.

Subtype Relation – Function Types

$$\boxed{\tau <: \tau}$$

Let's consider the subtype relation between **function types**.

```
val f = (g: Number => Top) => g(42)
```

Subtype Relation – Function Types

$$\boxed{\tau <: \tau}$$

Let's consider the subtype relation between **function types**.

```
val f = (g: Number => Top) => g(42)
```

```
// (Top => Top) <: (Number => Top)
val h: Top => Top = (x: Top) => x
f(h)
```

If the **parameter type** τ_1 is a **super type** of the **parameter type** τ_1', $\tau_1 \to \tau_2$ is a subtype of $\tau_1' \to \tau_2$.

Subtype Relation – Function Types

$$\boxed{\tau <: \tau}$$

Let's consider the subtype relation between **function types**.

```
val f = (g: Number => Top) => g(42)
```

```
// (Top => Top) <: (Number => Top)
val h: Top => Top = (x: Top) => x
f(h)
```

If the **parameter type** τ_1 is a **super type** of the **parameter type** τ_1', $\tau_1 \rightarrow \tau_2$ is a subtype of $\tau_1' \rightarrow \tau_2$.

```
// (Number => Number) <: (Number => Top)
val h: Number => Number = (x: Number) => x + 1
f(h)
```

Reversely, if the **return type** τ_2 is a **subtype** of the **return type** τ_2', $\tau_1 \rightarrow \tau_2$ is a subtype of $\tau_1 \rightarrow \tau_2'$.

Subtype Relation – Function Types

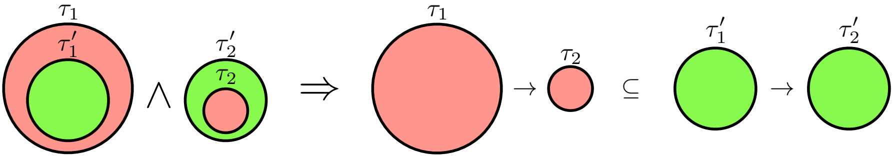
$$\boxed{\tau <: \tau}$$

If the following conditions are satisfied, $\tau_1 \to \tau_2$ is a subtype of $\tau_1' \to \tau_2'$.

- the **parameter type** τ_1 is a **super type** of the **parameter type** τ_1'.
- the **return type** τ_2 is a **subtype** of the **return type** τ_2'.

$$\frac{\tau_1 :> \tau_1' \qquad \tau_2 <: \tau_2'}{(\tau_1 \to \tau_2) <: (\tau_1' \to \tau_2')}$$

$$\tau_1 \to \tau_2 \qquad <: \qquad \tau_1' \to \tau_2'$$



$$\{f \mid \forall v \in \tau_1.\ f(v) \in \tau_2\} \qquad \{f \mid \forall v \in \tau_1'.\ f(v) \in \tau_2'\}$$

Typing Rules with Subsumption

One possible way to support subtype polymorphism is to add a general **subsumption** rule to the typing rules:

$$\frac{\Gamma \vdash e : \tau \qquad \tau <: \tau'}{\Gamma \vdash e : \tau'}$$

With this rule, we can type the following expression.

```
/* STFAE */
val x: {a: Number} = {a=2, b=3}; x.a
```

$$\dfrac{\dfrac{\dfrac{\cdots}{\varnothing \vdash \{a = 2, b = 3\} : \{a : \mathtt{num}, b : \mathtt{num}\}} \qquad \dfrac{\cdots}{\{a : \mathtt{num}, b : \mathtt{num}\} <: \{a : \mathtt{num}\}}}{\varnothing \vdash \{a = 2, b = 3\} : \{a : \mathtt{num}\}} \qquad \cdots}{\varnothing \vdash \mathtt{val}\ x : \{a : \mathtt{num}\} = \{a = 2, b = 3\};\ x.a : \mathtt{num}}$$

Typing Rules with Subsumption

One possible way to support subtype polymorphism is to add a general **subsumption** rule to the typing rules:

$$\frac{\Gamma \vdash e : \tau \qquad \tau <: \tau'}{\Gamma \vdash e : \tau'}$$

With this rule, we can type the following expression.

```
/* STFAE */
val x: {a: Number} = {a=2, b=3}; x.a
```

$$\dfrac{\dfrac{\dfrac{\cdots}{\varnothing \vdash \{a = 2, b = 3\} : \{a : \texttt{num}, b : \texttt{num}\}} \qquad \dfrac{\cdots}{\{a : \texttt{num}, b : \texttt{num}\} <: \{a : \texttt{num}\}}}{\varnothing \vdash \{a = 2, b = 3\} : \{a : \texttt{num}\}} \qquad \cdots}{\varnothing \vdash \texttt{val}\ x : \{a : \texttt{num}\} = \{a = 2, b = 3\};\ x.a : \texttt{num}}$$

However, it is **not algorithmic** because we don't know which types are required as the result of subsumption.

Algorithmic Typing Rules without Subsumption

Another way is to **directly apply** the subtype relation to the each typing rule without subsumption, and it is **algorithmic**.

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

$$\tau\text{-}\texttt{Add}\ \frac{\Gamma \vdash e_1 : \tau_1 \qquad \tau_1 <: \texttt{num} \qquad \Gamma \vdash e_2 : \tau_2 \qquad \tau_2 <: \texttt{num}}{\Gamma \vdash e_1 + e_2 : \texttt{num}}$$

$$\tau\text{-}\texttt{Mul}\ \frac{\Gamma \vdash e_1 : \tau_1 \qquad \tau_1 <: \texttt{num} \qquad \Gamma \vdash e_2 : \tau_2 \qquad \tau_2 <: \texttt{num}}{\Gamma \vdash e_1 \times e_2 : \texttt{num}}$$

$$\tau\text{-}\texttt{Val}_\tau\ \frac{\Gamma \vdash e_1 : \tau_1 \qquad \tau_1 <: \tau_0 \qquad \Gamma[x : \tau_0] \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{val}\ x : \tau_0\ \texttt{=}\ e_1;\ e_2 : \tau_2}$$

$$\tau\text{-}\texttt{App}\ \frac{\Gamma \vdash e_0 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_1 : \tau_3 \qquad \tau_3 <: \tau_1}{\Gamma \vdash e_0(e_1) : \tau_2}$$

Summary

1. Subtype Polymorphism
2. STFAE – TFAE with Subtype Polymorphism
   - Concrete Syntax
   - Abstract Syntax
3. Interpreter and Natural Semantics for STFAE
4. Type Checker and Typing Rules without Subtyping
   - Records
   - Field Accesses
   - Exit
   - Immutable Variable Definition
5. Subtype Relation
   - Bottom Type and Top Type
   - Record Types
   - Function Types
   - Typing Rules with Subsumption
   - Algorithmic Typing Rules without Subsumption

Exercise #15

https://github.com/ku-plrg-classroom/docs/tree/main/cose212/stfae

- 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.

Next Lecture

- Type Inference (1)

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