

Type Systems and Programming

D. Renault

ENSEIRB-Matmeca

Mar. 26th 2025, v.1.5.1

<https://www.labri.fr/perso/renault/working/teaching/stp/stp.php>

1 Subtyping

- Variance
- The contravariance curse
- Principles for variance

The subtyping test (1)

Rationale

- A value can have multiple types, and the types share inclusion relations.
- By definition, a value $v : T$ can also be typed $v : T'$ whenever $T \leq T'$.

Consequence :

▶ Substitution Lemma

$T \leq T'$ iff every value $v : T$ can be used in a context where T' is expected.

Subtyping test (example in Java)

Attempt to assign a value of type T into a variable of type T' unchanged.

```
Integer one      = 1;  
Number super_one = one; // OK
```

```
Number pi      = 3.14;  
Double sub_pi = pi; // Type error
```

The subtyping test (2)

Problem : implicit conversions

As a subtyping test, it has false positives, because many languages allow implicit conversions of values.

Example in C

Initializing an int variable with a float value forces an implicit conversion :

```
int main(void) {  
    int z = 3.14; // !!  
    z += 2.92;    // !!  
    printf("%d\n", z); }
```

```
% clang implicit.cpp -Wall -Wextra  
implicit.cpp:4:11: warning: implicit conversion  
from 'double' to 'int' changes value from 3.14 to 3  
    int z = 3.14;  
           ~   ^  
1 warning generated.
```

- The subtyping relation is supposed to be **antisymmetric**, but the implicit conversions blur this property.
- In this course, we consider subtyping without implicit conversions.

Summary on the different kinds of polymorphisms

- Each form of polymorphism defines new families of types, and these families share **subtyping relations**.
- The more families of types exist, the more **complex** the subtyping relation between types becomes.
- In this section, we explore some characteristics of this relation, and examine some implications in terms of programming.

Recall the subtyping theorems established for each form of polymorphism :

▶ **Lemma (Parametric subtyping) :**

Let $\forall T_1..T_n, U$ be a parametric type.

- **Parametric subtyping** : $\forall T_1..T_n, U <: [T_1 \mapsto U_1, .. T_n \mapsto U_n]U$.

▶ **Lemma (Record subtyping) :**

Let $T = \{l_i : T_i\}$ and $T' = \{m_i : T'_i\}$ be two record types.

- **Width subtyping** : if $T \supset T'$ as sets of pairs labels/types, then $T <: T'$.
- **Depth subtyping** : if T and T' share exactly the same labels and $\forall i, T_i <: T'_i$, then $T <: T'$.

▶ **Lemma (Intersection subtyping) :**

If T_1 and T_2 are two different types, $T_1 \& T_2 <: T_1$ and $T_1 \& T_2 <: T_2$

Where do subtyping relations come from ?

- Some relations are **structural**, meaning that they are related to the form of the underlying type families.
- Some relations may be deduced as **consequences** of existing subtyping relations.

Given for instance :

- the `List[·]` type as a type function $T \rightarrow \text{List}[T]$,
- two types `Banana` and `Fruit` such that `Banana <: Fruit`.

Can we deduce a subtyping relation between `List[Banana]` and `List[Fruit]` ?

For example : $\text{List}[\text{Banana}] \overset{?}{<} \text{List}[\text{Fruit}]$

\Rightarrow In other words, is the `List[·]` function increasing on types ?

Variance

Definition (Covariance / Contravariance / Invariance)

Let $f : \text{Typ} \rightarrow \text{Typ}$ be a function on types.

- f is said to be **covariant** iff it is increasing with regard to $<:$

$$\forall T, U, \quad T <: U \Rightarrow f(T) <: f(U)$$

- f is said to be **contravariant** iff it is decreasing with regard to $<:$

$$\forall T, U, \quad T <: U \Rightarrow f(T) >: f(U)$$

- If the images $f(T)$ and $f(U)$ are always incomparable when $T \neq U$, f is said to be **invariant**.

Example

The type of the immutable lists `List[·]` can be considered to be **covariant**.

- The Java generics are invariant, the variance appearing in wildcards.

```
// "? extends Number" refers to any subtype of Number  
ArrayList<? extends Number> a = new ArrayList<Integer>();
```

- C# support variance for generic interfaces, but the classes are invariant.

```
public interface IEnumerable<out T> { // "out T" indicates the covariance  
    public IEnumerable<T> Append<T> (IEnumerable<T> source, T elem); }
```


- Scala supports variance for generic interfaces and classes.

```
class List[+A] { // "+A" indicates the covariance  
    def append[B >: A](x : B) : List[B] } // append is called "::" in Scala
```

Function subtyping

Variance of the function type

The type $T \rightarrow U$ can be considered as a function (on types) of T and U . What kind of variance relations does it induce?

Recall  that subtyping can be thought of in terms of **substitution**.

Consider the following record type for storing Nat values :

$$S ::= \{\text{get} : \text{Unit} \rightarrow \text{Nat}, \text{set} : \text{Nat} \rightarrow \text{Unit}\}$$


Could such a store be replaced with one of the following types?

- $\{\text{get} : \text{Unit} \rightarrow \text{Int}, \text{set} : ..\}$ where $\text{Int} <: \text{Nat}$
- $\{\text{get} : \text{Unit} \rightarrow \text{Number}, \text{set} : ..\}$ where $\text{Nat} <: \text{Number}$

Function subtyping

Variance of the function type

The type $T \rightarrow U$ can be considered as a function (on types) of T and U . What kind of variance relations does it induce?

Recall  that subtyping can be thought of in terms of **substitution**.

Consider the following record type for storing Nat values :

$$S ::= \{\text{get} : \text{Unit} \rightarrow \text{Nat}, \text{set} : \text{Nat} \rightarrow \text{Unit}\}$$


Could such a store be replaced with one of the following types?

- $\{\text{get} : \text{Unit} \rightarrow \text{Int}, \text{set} : ..\}$ where $\text{Int} <: \text{Nat}$ 
- $\{\text{get} : \text{Unit} \rightarrow \text{Number}, \text{set} : ..\}$ where $\text{Nat} <: \text{Number}$ 

Function subtyping

Variance of the function type

The type $T \rightarrow U$ can be considered as a function (on types) of T and U . What kind of variance relations does it induce?

Recall  that subtyping can be thought of in terms of **substitution**.

Consider the following record type for storing Nat values :

$$S ::= \{\text{get} : \text{Unit} \rightarrow \text{Nat}, \text{set} : \text{Nat} \rightarrow \text{Unit}\}$$


Could such a store be replaced with one of the following types?

- $\{\text{get} : \dots, \text{set} : \text{Int} \rightarrow \text{Unit}\}$ where $\text{Int} <: \text{Nat}$
- $\{\text{get} : \dots, \text{set} : \text{Number} \rightarrow \text{Unit}\}$ where $\text{Nat} <: \text{Number}$

Function subtyping

Variance of the function type

The type $T \rightarrow U$ can be considered as a function (on types) of T and U . What kind of variance relations does it induce?

Recall  that subtyping can be thought of in terms of **substitution**.

Consider the following record type for storing Nat values :

$$S ::= \{\text{get} : \text{Unit} \rightarrow \text{Nat}, \text{set} : \text{Nat} \rightarrow \text{Unit}\}$$

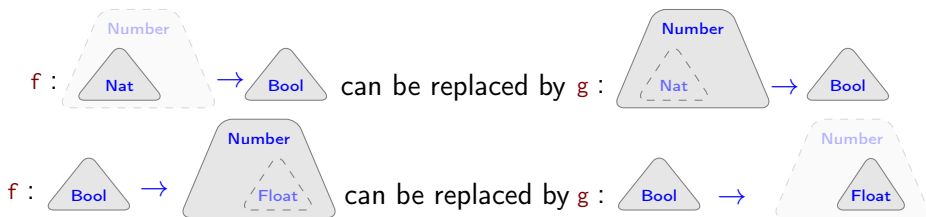
Could such a store be replaced with one of the following types?

- $\{\text{get} : \dots, \text{set} : \text{Int} \rightarrow \text{Unit}\}$ where $\text{Int} <: \text{Nat}$ 
- $\{\text{get} : \dots, \text{set} : \text{Number} \rightarrow \text{Unit}\}$ where $\text{Nat} <: \text{Number}$ 

Lemma (Function subtyping) :

A function type is **covariant** in its **result type** and **contravariant** in its **parameter type**.

If $T_{inf} <: T_{sup}$ and $U_{inf} <: U_{sup}$, then $(T_{sup} \rightarrow U_{inf}) <: (T_{inf} \rightarrow U_{sup})$



The contravariance curse

Subtyping is often used as a means to refine types (e.g. with inheritance). But the variance rules somewhat hinder these forms of refinements.

- Consider the example for a record type that compares numbers :

$$\text{EqNumber} ::= \{\text{val} : \text{Number}, \text{equal} : \text{Number} \rightarrow \text{Bool}\}$$

- Consider now inheriting from this with more precise internal numbers :

$$\text{EqFloat} ::= \{\text{val} : \text{Float}, \text{equal} : \text{Float} \rightarrow \text{Bool}\}$$

Deceptive (but also disappointing) fact

$$\text{EqFloat} \not\leq \text{EqNumber}$$

```

class Point {
    int x, y;
    Point(int _x, int _y)    { x = _x; y = _y; }
    boolean equal(Point other) { return (other.x == x) &&
                                     (other.y == y); }
}

class ColPoint extends Point {
    int c;
    ColPoint(int _x, int _y, int _c) { super(_x, _y); c = _c; }
    boolean equal(ColPoint other)    { return (other.c == c) &&
                                     (super.equals(other)); }
}

static boolean isOrigin(Point p) { return p.equal(new ColPoint(0,0,0)); }

```

- What happens when calling `isOrigin(new ColPoint(0,0,7))`?

```

class Point {
    int x, y;
    Point(int _x, int _y) { x = _x; y = _y; }
    boolean equal(Point other) { return (other.x == x) &&
                                     (other.y == y); }
}

class ColPoint extends Point {
    int c;
    ColPoint(int _x, int _y, int _c) { super(_x, _y); c = _c; }
    boolean equal(ColPoint other) { return (other.c == c) &&
                                             (super.equals(other)); }
}

static boolean isOrigin(Point p) { return p.equal(new ColPoint(0,0,0)); }

```

- What happens when calling `isOrigin(new ColPoint(0,0,7))` ?
 - `ColPoint.equal` ✗ `Point.equal` because of the contravariance rule.

```

class Point {
    int x, y;
    Point(int _x, int _y)    { x = _x; y = _y; }
    boolean equal(Point other) { return (other.x == x) &&
                                   (other.y == y); }
}

class ColPoint extends Point {
    int c;
    ColPoint(int _x, int _y, int _c) { super(_x, _y); c = _c; }
    boolean equal(ColPoint other)    { return (other.c == c) &&
                                   (super.equals(other)); }
}

static boolean isOrigin(Point p) { return p.equal(new ColPoint(0,0,0)); }

```

- What happens when calling `isOrigin(new ColPoint(0,0,7))` ?
 - `ColPoint.equal` ✗ `Point.equal` because of the contravariance rule.
 - `equal` is overloaded, it uses `Points` in `isOrigin` and returns `true`.

```

class Point {
    int x, y;
    Point(int _x, int _y)    { x = _x; y = _y; }
    boolean equal(Point other) { return (other.x == x) &&
                                     (other.y == y); }
}

class ColPoint extends Point {
    int c;
    ColPoint(int _x, int _y, int _c) { super(_x, _y); c = _c; }
    boolean equal(ColPoint other)    { return (other.c == c) &&
                                     (super.equals(other)); }
}

static boolean isOriginBis(Point p) { return p.equal(new Point(0,0)); }

```

- What happens if we suppose that `ColPoint.equal` <: `Point.equal` anyway?

```

class Point {
    int x, y;
    Point(int _x, int _y)    { x = _x; y = _y; }
    boolean equal(Point other) { return (other.x == x) &&
                                     (other.y == y); }
}

class ColPoint extends Point {
    int c;
    ColPoint(int _x, int _y, int _c) { super(_x, _y); c = _c; }
    boolean equal(ColPoint other)    { return (other.c == c) &&
                                     (super.equals(other)); }
}

static boolean isOriginBis(Point p) { return p.equal(new Point(0,0)); }

```

- What happens if we suppose that `ColPoint.equal <: Point.equal` anyway?
 - the call `isOriginBis(new ColPoint(0,0,7))` is well-typed,

```

class Point {
    int x, y;
    Point(int _x, int _y)    { x = _x; y = _y; }
    boolean equal(Point other) { return (other.x == x) &&
                                   (other.y == y); }
}

class ColPoint extends Point {
    int c;
    ColPoint(int _x, int _y, int _c) { super(_x, _y); c = _c; }
    boolean equal(ColPoint other)    { return (other.c == c) &&
                                   (super.equals(other)); }
}

static boolean isOriginBis(Point p) { return p.equal(new Point(0,0)); }

```

- What happens if we suppose that `ColPoint.equal` <: `Point.equal` anyway?
 - the call `isOriginBis(new ColPoint(0,0,7))` is well-typed,
 - it uses the `ColPoint` equality with a `Point` parameter, and gets stuck.

Principles for variance of type variables

A type playing the role of a **supplier** can vary **covariantly** and must **not** vary contravariantly.

For example : r-values, getters, results of functions

A type playing the role of a **receiver** can vary **contravariantly** and must **not** vary covariantly.

For example : l-values, setters, parameters of functions

In Java, this takes the form of the “**Get and Put principle**” popularized by Naftalin and Wadler in *Java Generics* (2006) :

*“Use an **extends** wildcard when you only **get** values out of a structure, use a **super** wildcard when you only **put** values into a structure, and don't use a wildcard when you both get and put.”*

Consider two types `ColPoint <: Point`, and the following generic interfaces :

<code>GetF[T]</code>	<code>::= {get : Unit → T}</code>	covariant	contravariant
<code>SetF[T]</code>	<code>::= {set : T → Unit}</code>	contravariant	covariant

- The following examples are unsafe if the variance is reversed :

```
let gpt : GetF[Point] = ... in
(* Valid if GetF[.] is contravariant *)
let gcpt : GetF[ColPoint] = gpt in
(* Unsafe access to inexistant color field *)
gcpt.get().color
```

```
let gcpt : SetF[ColPoint] = ... in
(* Valid if SetF[.] is covariant *)
let gpt : SetF[Point] = gcpt in
(* Unsafe setting of missing color field *)
gpt.set(new Point())
```

- The following is an example of type unsafety in Java :

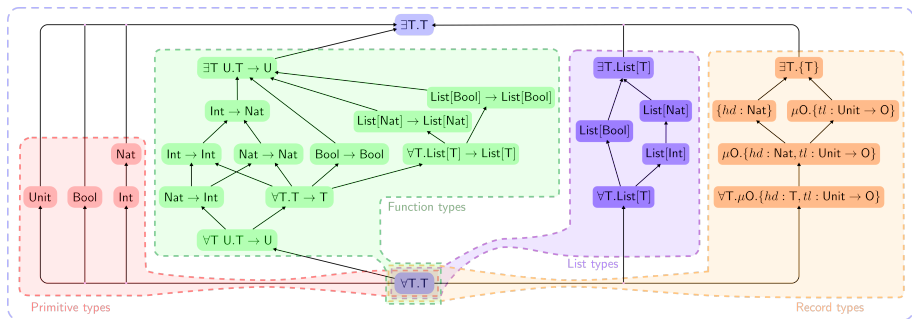
```
String[] strings = new String[1];
Object[] objects = strings;           // Arrays are covariant
objects[0] = new Integer(1);          // Runtime failure
```

The Java arrays implement both `GetF` and `SetF`, and should be invariant.

Summary on subtyping

- Subtyping is an important property of a type system, enabling to use the polymorphism at full strength.
- The subtyping relations are refined by the notion of **variance**.
- Many constructions of the language (functions, records ...) are the source of specific subtyping rules.
It is also possible to refine these relations by defining variance relations for particular type functions.
- Depending on the **coherence** of the relations, **type unsafety** problems may appear when programming.
- Deciding the subtyping relation also relies on verifying the coherence of the different local subtype relations.

The type lattice



Deciding the subtype relation (1)

In order to deal with object types, it is necessary to decide the subtyping relation on a wide family of types, among which recursive types.

Subtyping decision algorithm

The function `subtype`(\mathcal{A} , S , T) decides whether $S <: T$.

- It considers a set of assumptions \mathcal{A} , each assumption being a pair (S_i, T_i) such that $S_i <: T_i$. Initially, the set is empty.
- Depending on the form of S and T , it deduces new assumptions.
- It terminates either when finding an incoherent set of assumptions or returns a set of coherent assumptions.

Basically, `subtype` builds a subset of the type lattice.

Definition (Subtyping decision algorithm)

The function $\text{subtype}(\mathcal{A}, S, T)$ is defined as :

- if $S = T$ or $(S, T) \in \mathcal{A}$, return \mathcal{A}
- if $(T, S) \in \mathcal{A}$, fail
- else let $\mathcal{A}_0 = \mathcal{A} \cup (S, T)$ and depending on (S, T) :
 - $S = \{l_i : S_i\}_{i=[1..n+m]}$ and $T = \{l_i : T_i\}_{i=[1..n]}$, then compute in sequence $\mathcal{A}_i = \text{subtype}(\mathcal{A}_{i-1}, S_i, T_i)$ for $i \in [1..n]$, return fail if any computation fails otherwise return \mathcal{A}_n .
 - $S = S_1 \rightarrow S_2$ and $T = T_1 \rightarrow T_2$ then let $\mathcal{A}_1 = \text{subtype}(\mathcal{A}_0, T_1, S_1)$ and $\mathcal{A}_2 = \text{subtype}(\mathcal{A}_1, S_2, T_2)$ in return fail if any computation fails otherwise return \mathcal{A}_2 .
 - $T = \mu X. T_1$ then compute $\text{subtype}(\mathcal{A}_0, S, [X \mapsto \mu X. T_1] T_1)$
 - $S = \mu X. S_1$ then compute $\text{subtype}(\mathcal{A}_0, [X \mapsto \mu X. S_1] S_1, T)$
 - otherwise fail.

Deciding the subtype relation (2)

Consider the following two object types :

$$\begin{aligned} O &::= \mu X. \{\text{clone} : \text{Unit} \rightarrow X\} \\ S &::= \mu Y. \{\text{clone} : \text{Unit} \rightarrow Y, \text{val} : \text{Nat}\} \end{aligned}$$

The computation of $\text{subtype}(S, O)$ passes through the following steps :

- $\text{subtype}(S, O)$
- $\text{subtype}(S, \{\text{clone} : \text{Unit} \rightarrow O\})$
- $\text{subtype}(\{\text{clone} : \text{Unit} \rightarrow S, \text{val} : \text{Nat}\}, \{\text{clone} : \text{Unit} \rightarrow O\})$
- $\text{subtype}(\text{Unit} \rightarrow S, \text{Unit} \rightarrow O)$
- $\text{subtype}(\text{Unit}, \text{Unit})$ and $\text{subtype}(S, O)$ then terminates successfully.

As a result, this proves $S <: O$, even if they both are recursive types.

- Pierce, B. C. *Types and Programming Languages*. MIT Press, 2002.
- Bruce, K. B. *Foundations of Object-oriented Languages : Types and Semantics*. MIT Press, 2002.
- Hindley, J. R. *Basic simple type theory*. Cambridge University Press, 1997.
- Wadler, P. *Propositions as types*. Communications ACM, 2015.