# Kotlin language specification

Version 1.9-rfc+0.1

Marat Akhin Mikhail Belyaev

# Chapter 13

# Kotlin type constraints

Some complex tasks that need to be solved when compiling Kotlin code are formulated best using *constraint systems* over Kotlin types. These are solved using constraint solvers.

## 13.1 Type constraint definition

A *type constraint* in general is an inequation of the following form: $T <: U$ where $T$ and $U$ are concrete Kotlin types. As Kotlin has parameterized types, $T$ and $U$ may be *free type variables*: unknown types which may be substituted by any other type in Kotlin.

Please note that, in general, not all type parameters are considered as free type variables in a constraint system. Some type variables may be *fixed* in a constraint system; for example, type parameters of a parameterized class *inside* its body are unknown types, but are not free type variables either. A fixed type variable describes an unknown, but fixed type which is *not* to be substituted.

We will use the notation $T_i$ for a type variable and $\tilde{T}_i$ for a fixed type variable. The main difference between fixed type variables and concrete types is that different concrete types may not be equal, but a fixed type variable may be equal to another fixed type variable or a concrete type.

> Examples of valid type constraints:
>
> - `List<`$\widetilde{X}$`>` $<:$ $Y$
> - `List<`$\widetilde{X}$`>` $<:$ `List<List<Int>>`
> - $\widetilde{X}$ $<:$ $Y$

Every constraint system has general implicit constraints $T_j <:$ `kotlin.Any?` and `kotlin.Nothing` $<: T_j$ for every type $T_j$ mentioned in the system, including type variables.

## 13.2 Type constraint solving

There are two tasks which a type constraint solver may perform: checking constraint system for soundness, i.e., if a solution exists, and solving constraint system, i.e., inferring a satisfying substitution of concrete types for all free type variables.

Checking a constraint system for soundness can be viewed as a much simpler case of solving that constraint system: if there is a solution, the system is sound, meaning there are only two possible outcomes. Solving a constraint system, on the other hand, may have multiple possible outcomes, as there may be multiple valid solutions.

> Example: constraint systems which are sound yet no relevant solutions exist.
>
> - $X <: Y$
> - `List<`$X$`>` $<:$ `Collection<`$X$`>`

### 13.2.1 Checking constraint system soundness

Checking constraint system soundness is a satisfiability problem. That is, given a number of constraints in the form $S <: T$ containing zero or more free type variables (also called *inference type variables*), it needs to determine if these constraints are non-contradictory, i.e., if there exists a possible instantiation of these free variables to concrete types which makes all given constraints valid.

This problem can be reduced to finding a set of lower and upper bounds for each of these variables and determining if these bounds are non-contradictory. The algorithm of finding these bounds is implementation-defined and is not guaranteed to prove the satisfiability of given constraints in all possible cases.

#### A sample bound inference algorithm

The algorithm given in this section is just an example of a family of algorithms that may be applied to the problem given above. A particular implementation is not guaranteed to follow this algorithm, but one may use it as a reference on how this problem may be approached.

> Note: a well-informed reader may notice this algorithm to be similar to the one used by Java. This is not a coincidence: our sample inference algorithm has indeed been inspired by Java's.

The algorithm works in two phases: *reduction* and *incorporation* which are applied to the constraint system and its current solution in turns until a fixpoint or an error is reached (aka reduction-incorporation procedure or RIP). The reduction phase is used to produce bounds for inference variables based on constraints; this phase is also responsible for eliminating the constraints which

are no longer needed. The incorporation phase is used to introduce new bounds and constraints from existing bounds.

A bound is similar to a constraint in that it has the form $S <: T$, at least one of $S$ or $T$ is an inference variable. Thus, the current (and also the final) solution is a set of upper and lower bounds for each inference variable. A *resolved type* in this context is any type which does not contain inference variables.

**Reduction phase**: for each constraint $S <: T$ in the constraint system the following rules are applied:

- If $S$ and $T$ are resolved types and:
  - If $S <: T$, this constraint is eliminated;
  - Otherwise, this is an inference error;
- Otherwise, if $S$ is an inference variable $\alpha$, a new bound $\alpha <: T$ is added to current solution;
- Otherwise, if $T$ is an inference variable $\beta$, a new bound $S <: \beta$ is added to current solution;
- Otherwise, if $S$ is a flexible type of the form $(\alpha..\alpha?)$ where $\alpha$ is an inference variable, a new bound $\alpha <: (T..T?)$ is added to current solution;
- Otherwise, if $T$ is a flexible type of the form $(\alpha..\alpha?)$ where $\alpha$ is an inference variable, a new bound $(S..S?) <: \alpha$ is added to current solution;
- Otherwise, if $S$ is a nullable type of the form $A?$ and:
  - If $T$ is a known non-nullable type (a classifier type, a nullability-asserted type $B!!$, a type variable with a known non-nullable lower bound, or an intersection type containing a known non-nullable type), this is an inference error;
  - Otherwise, the constraint is reduced to $A <: T$. Also, if $T$ is also a nullable type of the form $B?$, an additional constraint $A!! <: B$ is introduced;
- Otherwise, if $S$ is a flexible type of the form $(B..A?)$ and:
  - If $T$ is a nullable type of form $C?$, the constraint is reduced to $(B..A) <: C$, or to $A <: C$ if $A \equiv B$;
  - Otherwise, the constraint is reduced to $(B..A) <: T$, or to $A <: T$ if $A \equiv B$;
- Otherwise, if $T$ is a parameterized type $G[A_1, \ldots, A_N]$, among all supertypes of $S$ the one of the form $G[B_1, \ldots, B_N]$ is chosen.
  - If no such supertype exists, this is an inference error;
  - Otherwise, for each $M \in [1, N]$, a type argument constraint for containment $A_M \preceq B_M$ is introduced (see below);
- Otherwise, if $T$ is any other classifier type and $T$ is among supertypes for $S$, the constraint is eliminated; otherwise, this is an inference error;
- Otherwise, if $T$ is a type variable and:
  - If $S$ is an intersection type containing $T$, this constraint is eliminated;
  - Otherwise, if $T$ has a lower bound $B$, the constraint is reduced to $S <: B$;
  - Otherwise, this is an inference error;

- Otherwise, if $T$ is an intersection type $A_1 \,\&\, \ldots \,\&\, A_N$, the constraint is reduced to $N$ constraints $S <: A_M$ for each $M \in [1, N]$;
- Otherwise, if $T$ is a nullable type of the form $B?$ and:
  - If $S$ is a known non-nullable type (a classifier type, a nullability-asserted type $A!!$, a type variable with a known non-nullable lower bound, or an intersection type containing a known non-nullable type), the constraint is reduced to $S <: B$;
  - Otherwise, this is an inference error.

Type argument constraints for a containment relation $Q \preceq F$ are constructed as follows:

> Important: for the purposes of this algorithm, declaration-site variance type arguments are considered to be their equivalent use-site variance versions.

- If either $Q$ or $F$ is a special bivariant type argument $\star$, no constraints are produced;
- If $F$ has the form $F'$ (is invariant):
  - If $Q$ is also invariant and of the form $Q'$, two constraints are produced: $F' <: Q'$ and $Q' <: F'$;
  - If $Q$ has any other variance, this is an inference error;
- If $F$ has the form `out` $F'$ (is covariant):
  - If $Q$ has the form `out` $Q'$ or $Q'$, the following constraint is produced: $Q' <: F'$;
  - If $Q$ has the form `in` $Q'$, the following constraint is produced: `kotlin.Any?` $<: F'$;
- If $F$ has the form `in` $F'$ (is contravariant):
  - If $Q$ has the form `in` $Q'$ or $Q'$, the following constraint is produced: $F' <: Q'$;
  - If $Q$ has the form `out` $Q'$, the following constraint is produced: $F' <:$ `kotlin.Nothing`.

**Incorporation phase**: for each bound and particular bound combinations in the current solution, new constraints are produced as follows (it is safe to assume that each constraint is introduced into the system only once, so if this step produces constraints that have already been reduced, they are not added into the system):

- For each inference variable $\alpha$, for each pair of bounds $S <: \alpha$ and $\alpha <: T$, a new constraint is produced: $S <: T$;
- For each inference variable $\alpha$, if there is a pair of bounds $S <: \alpha$ and $\alpha <: S$ (i.e., $\alpha$ is equivalent to $S$), for each bound $Q <: P$ where $Q$ or $P$ contains $\alpha$, a new constraint is produced: $Q[\alpha := S] <: P[\alpha := S]$;
- For each inference variable $\alpha$, for each pair of bounds $\alpha <: S$ and $\alpha <: T$ where $S$ has a supertype of the form $G[A_1, \ldots, A_N]$ and $T$ has a matching supertype of the form $G[B_1, \ldots, B_N]$, for each matching supertype $G$ and each $M \in [1, N]$, if both $A_M$ and $B_M$ are invariant and have forms $A'_M$ and

$B'_M$ respectively, the following new constraints are produced: $A'_M <: B'_M$ and $B'_M <: A'_M$.

### 13.2.2 Finding optimal constraint system solution

As any constraint system may have multiple valid solutions, finding one which is "optimal" in some sense is not possible in general, because the notion of the best solution for a task depends on the said task. To deal with this, a constraint system allows two additional types of constraints:

- A *pull-up* constraint for type variable $T$, denoted $\uparrow T$, signifying that when finding a substitution for this variable, the optimal solution is the largest one according to subtyping relation;
- A *push-down* constraint for type variable $T$, denoted $\downarrow T$, signifying that when finding a substitution for this variable, the optimal solution is the smallest one according to subtyping relation.

If a variable has no constraints of these kinds associated with it, it is assumed to have a pull-up implicit constraint. The process of instantiating the free variables of a constraint system starts by finding the bounds for each free variable (as mentioned in the previous section) and then, given these bounds, continues to pick the right type from them. Excluding other free variables, this boils down to:

- For a variable with a push-down constraint, the solution is the greatest lower bound of all upper bounds for this variable, excluding other free variables;
- For a variable with a pull-up constraint, the solution is the least upper bound of all lower bounds for this variable, excluding other free variables;
- For a variable with both or none, the solution is also the least upper bound of all lower bounds for this variable, excluding other free variables.

If there are inference variables dependent on other inference variables ($\alpha$ is dependent on $\beta$ iff there is a bound $\alpha <: T$ or $T <: \alpha$ where $T$ contains $\beta$), this process is performed in stages.

During each stage a set of inference variables not dependent on other inference variables (but possibly dependent on each other) is selected, the solutions for these variables are found using existing bounds, and after that these variables are **resolved** in the current bound set by replacing all of their instances in other bounds by the solution. This may trigger a new RIP.

After that, a new independent set of inference variables is picked and this process is repeated until an inference error occurs or a solution for each inference variable is found.

### 13.2.3 The relations on types as constraints

In other sections (for example, Expressions and Statements) the relations between types may be expressed using the type operations found in the type system

section of this document.

The greatest lower bound of two types is converted directly as-is, as the greatest lower bound is always an intersection type.

The least upper bound of two types is converted as follows. If type $T$ is defined to be the least upper bound of $A$ and $B$, the following constraints are produced:

- $A <: T$
- $B <: T$
- $\downarrow T$
- $\uparrow A$
- $\uparrow B$

  Important: the results of finding GLB or LUB via a constraint system may be different from the results of finding them via a normalization procedure (i.e., imprecise); however, they are sound w.r.t. bound, meaning a constraint system GLB is still a lower bound and a constraint system LUB is still an upper bound.

  Example:

  Let's assume we have the following code:

  ```
  val e = if (c) a else b
  ```

  where `a`, `b`, `c` are some expressions with unknown types (having no other type constraints besides the implicit ones).

  Assume the type variables generated for them are $A$, $B$ and $C$ respectively, the type variable for `e` is $E$. According to the conditional expression rules, this produces the following relations:

  - $C <:$ `kotlin.Boolean`
  - $E = \text{LUB}(A, B)$

  These, in turn, produce the following explicit constraints:

  - $C <:$ `kotlin.Boolean`
  - $A <: E$
  - $B <: E$
  - $\downarrow E$
  - $\uparrow A$
  - $\uparrow B$

  which, w.r.t. general and pull-up implicit constraints, produce the following solution:

  - $C \rightarrow$ `kotlin.Boolean`
  - $A \rightarrow$ `kotlin.Any?`
  - $B \rightarrow$ `kotlin.Any?`
  - $E \rightarrow$ `kotlin.Any?`