Kotlin language specification

Version 1.9-rfc+0.1

Marat Akhin Mikhail Belyaev

Chapter 14

Type inference

Kotlin has a concept of *type inference* for compile-time type information, meaning some type information in the code may be omitted, to be inferred by the compiler. There are two kinds of type inference supported by Kotlin.

- Local type inference, for inferring types of expressions locally, in statement/expression scope;
- Function signature type inference, for inferring types of function return values and/or parameters.

Type inference is a type constraint problem, and is usually solved by a type constraint solver. For this reason, type inference is applicable in situations when the type context contains enough information for the type constraint solver to create an optimal constraint system solution w.r.t. type inference problem.

> Note: for the purposes of type inference, an optimal solution is the one which does not contain any free type variables with no explicit constraints on them.

Kotlin also supports flow-sensitive types in the form of smart casts, which have direct effect on type inference. Therefore, we will discuss them first, before talking about type inference itself.

14.1 Smart casts

Kotlin introduces a limited form of flow-sensitive typing called *smart casts*. Flow-sensitive typing means some expressions in the program may introduce changes to the compile-time types of variables. This allows one to avoid unneeded explicit casting of values in cases when their runtime types are guaranteed to conform to the expected compile-time types.

Flow-sensitive typing may be considered a specific instance of traditional data-flow analysis. Therefore, before we discuss it further, we need to establish the data-flow framework, which we will use for smart casts.

14.1.1 Data-flow framework

Smart cast lattices

We assume our data-flow analysis is run on a classic control-flow graph (CFG) structure, where most non-trivial expressions and statements are simplified and/or desugared.

Our data-flow domain is a map lattice SmartCastData = Expression $\to$ SmartCastType, where Expression is any Kotlin expression and SmartCastType = Type $\times$ Type sublattice is a product lattice of smart cast data-flow facts of the following kind.

- First component describes the type, which an expression definitely **has**
- Second component describes the type, which an expression definitely **does not have**

The sublattice order, join and meet are defined as follows.

$$P_1 \times N_1 \sqsubseteq P_2 \times N_2 \Leftrightarrow P_1 <: P_2 \wedge N_1 :> N_2$$

$$P_1 \times N_1 \sqcup P_2 \times N_2 = \mathrm{LUB}(P_1, P_2) \times \mathrm{GLB}(N_1, N_2)$$
$$P_1 \times N_1 \sqcap P_2 \times N_2 = \mathrm{GLB}(P_1, P_2) \times \mathrm{LUB}(N_1, N_2)$$

Note: a well-informed reader may notice the second component is behaving very similarly to a *negation* type.

$$\begin{aligned}(P_1 \,\&\, \neg N_1) \mid (P_2 \,\&\, \neg N_2) &\sqsubseteq (P_1 \mid P_2) \,\&\, (\neg N_1 \mid \neg N_2) \\ &= (P_1 \mid P_2) \,\&\, \neg(N_1 \,\&\, N_2) \\ (P_1 \,\&\, \neg N_1) \,\&\, (P_2 \,\&\, \neg N_2) &= (P_1 \,\&\, P_2) \,\&\, (\neg N_1 \,\&\, \neg N_2) \\ &= (P_1 \,\&\, P_2) \,\&\, \neg(N_1 \mid N_2)\end{aligned}$$

This is as intended, as "type which an expression definitely does not have" is exactly a negation type. In smart casts, as Kotlin type system does not have negation types, we overapproximate them when needed.

Smart cast transfer functions

The data-flow information uses the following transfer functions.

$$
\begin{aligned}
[\![assume(x\ \texttt{is}\ T)]\!]\,(s) &= s[x \to s(x) \sqcap (T \times \top)] \\
[\![assume(x\ \texttt{!is}\ T)]\!]\,(s) &= s[x \to s(x) \sqcap (\top \times T)] \\
\\
[\![x\ \texttt{as}\ T]\!]\,(s) &= s[x \to s(x) \sqcap (T \times \top)] \\
[\![x\ \texttt{!as}\ T)]\!]\,(s) &= s[x \to s(x) \sqcap (\top \times T)] \\
\\
[\![assume(x == null)]\!]\,(s) &= s[x \to s(x) \sqcap (\texttt{kotlin.Nothing?} \times \top)] \\
[\![assume(x \texttt{ != } null)]\!]\,(s) &= s[x \to s(x) \sqcap (\top \times \texttt{kotlin.Nothing?})] \\
\\
[\![assume(x === null)]\!]\,(s) &= s[x \to s(x) \sqcap (\texttt{kotlin.Nothing?} \times \top)] \\
[\![assume(x \texttt{ !== } null)]\!]\,(s) &= s[x \to s(x) \sqcap (\top \times \texttt{kotlin.Nothing?})] \\
\\
[\![assume(x == y)]\!]\,(s) &= s[x \to s(x) \sqcap s(y), \\
&\qquad y \to s(x) \sqcap s(y)] \\
[\![assume(x \texttt{ != } y)]\!]\,(s) &= s[x \to s(x) \sqcap swap(isNullable(s(y))), \\
&\qquad y \to s(y) \sqcap swap(isNullable(s(x)))] \\
\\
[\![assume(x === y)]\!]\,(s) &= s[x \to s(x) \sqcap s(y), \\
&\qquad y \to s(x) \sqcap s(y)] \\
[\![assume(x \texttt{ !== } y)]\!]\,(s) &= s[x \to s(x) \sqcap swap(isNullable(s(y))), \\
&\qquad y \to s(y) \sqcap swap(isNullable(s(x)))] \\
\\
[\![x = y]\!]\,(s) &= s[x \to s(y)] \\
\\
[\![killDataFlow(x)]\!]\,(s) &= s[x \to (\top \times \top)] \\
\\
[\![l]\!]\,(s) &= \bigsqcup_{p \in predecessor(l)} [\![p]\!]\,(s)
\end{aligned}
$$

where

$$swap(P \times N) = N \times P$$

$$isNullable(s) = \begin{cases} (\texttt{kotlin.Nothing?} \times \top) & \text{if } s \sqsubseteq (\texttt{kotlin.Nothing?} \times \top) \\ (\top \times \top) & \text{otherwise} \end{cases}$$

Important: transfer functions for `==` and `!=` are used only if the corresponding `equals` implementation is known to be equivalent to

reference equality check. For example, generated `equals` implementation for data classes is considered to be equivalent to reference equality check.

Note: in some cases, after the CFG simplification a program location l may be duplicated and associated with several locations $l_1, \ldots, l_N$ in the resulting CFG. If so, the data-flow information for l is calculated as

$$[\![l]\!] = \bigsqcup_{i=1}^{N} [\![l_i]\!]$$

Note: a *killDataFlow* instruction is used to reset the data-flow information in cases, when a compiler deems necessary to stop its propagation. For example, it may be used in loops to speed up data-flow analysis convergence. This is the current behaviour of the Kotlin compiler.

After the data-flow analysis is done, for a program location l we have its data-flow information $[\![l]\!]$, which contains data-flow facts $[\![l]\!]\,[e] = (P \times N)$ for an expression e.

14.1.2 Smart cast types

The data-flow information is used to produce the smart cast type as follows.

First, smart casts may influence the compile-time type of an expression e (called *smart cast sink*) only if the sink is *stable*.

Second, for a stable smart cast sink e we calculate the overapproximation of its possible type.

$$[\![l]\!]\,[e] = (P \times N) \Rightarrow \mathit{smartCastTypeOf}(e) = \mathit{typeOf}(e) \& P \& \mathit{approxNegationType}(N)$$

$$\mathit{approxNegationType}(N) = \begin{cases} \texttt{kotlin.Any} & \text{if } \texttt{kotlin.Nothing?} <: N \\ \texttt{kotlin.Any?} & \text{otherwise} \end{cases}$$

As a result, $\mathit{smartCastTypeOf}(e)$ is used as a compile-time type of e for most purposes (including, but not limited to, function overloading and type inference of other values).

Note: the most important exception to when smart casts are used in type inference is direct property declaration.

```
fun noSmartCastInInference() {
    var a: Any? = null

    if (a == null) return

    var c = a // Direct property declaration

    c // Declared type of `c` is Any?
      // However, here it's smart casted to Any
}

fun <T> id(a: T): T = a

fun smartCastInInference() {
    var a: Any? = null

    if (a == null) return

    var c = id(a)

    c // Declared type of `c` is Any
}
```

Smart casts are introduced by the following Kotlin constructions.

- Conditional expressions (`if`)
- When expressions (`when`);
- Elvis operator (operator `?:`);
- Safe navigation operator (operator `?.`);
- Logical conjunction expressions (operator `&&`);
- Logical disjunction expressions (operator `||`);
- Not-null assertion expressions (operator `!!`);
- Cast expressions (operator `as`);
- Type-checking expressions (operator `is`);
- Simple assignments;
- Platform-specific cases: different platforms may add other kinds of expressions which introduce additional smart cast sources.

  Note: property declarations are not listed here, as their types are derived from initializers.

  Note: for the purposes of smart casts, most of these constructions are simplified and/or desugared, when we are building the program CFG for the data-flow analysis. We informally call such constructions *smart cast sources*, as they are responsible for creating smart cast specific instructions.

14.1.3 Smart cast sink stability

A smart cast sink is *stable* for smart casting if its value cannot be changed via means external to the CFG; this guarantees the smart cast conditions calculated by the data-flow analysis still hold at the sink. This is one of the necessary conditions for smart cast to be applicable to an expression.

Smart cast sink stability breaks in the presence of the following aspects.

- concurrent writes;
- mutable value capturing;
- separate module compilation;
- custom getters;
- delegation.

The following smart cast sinks are considered stable.

1. Immutable local or classifier-scope properties without delegation or custom getters;
2. Mutable local properties without delegation or custom getters, if the compiler can prove that they are effectively immutable, i.e., cannot be changed by external means;
3. Immutable properties of immutable stable properties without delegation or custom getters, if they are declared in the current module.

Effectively immutable smart cast sinks

We will call redefinition of e ***direct*** redefinition, if it happens in the same declaration scope as the definition of e. If e is redefined in a nested declaration scope (w.r.t. its definition), this is a ***nested*** redefinition.

> Note: informally, a nested redefinition means the property has been captured in another scope and may be changed from that scope in a concurrent fashion.

We define ***direct*** and ***nested*** smart cast sinks in a similar way.

> Example:

```
fun example() {
    // definition
    var x: Int? = null

    if (x != null) {
        run {
            // nested smart cast sink
            x.inc()

            // nested redefinition
            x = ...
```

```
        }
        // direct smart cast sink
        x.inc()
    }

    // direct redefinition
    x = ...
}
```

A mutable local property P defined at D is considered effectively immutable at a direct sink S, if there are no nested redefinitions on any CFG path between D and S.

A mutable local property P defined at D is considered effectively immutable at a nested sink S, if there are no nested redefinitions of P and all direct redefinitions of P precede S in the CFG.

Example:

```
fun directSinkOk() {
    var x: Int? = 42 // definition
    if (x != null)   // smart cast source
        x.inc()      // direct sink
    run {
        x = null     // nested redefinition
    }
}

fun directSinkBad() {
    var x: Int? = 42 // definition
    run {
        x = null     // nested redefinition
                     //   between a definition
                     //   and a sink
    }
    if (x != null)   // smart cast source
        x.inc()      // direct sink
}

fun nestedSinkOk() {
    var x: Int? = 42     // definition
    x = getNullableInt() // direct redefinition
    run {
        if (x != null)   // smart cast source
            x.inc()      // nested sink
    }
}
```

```
fun nestedSinkBad01() {
    var x: Int? = 42      // definition
    run {
        if (x != null)    // smart cast source
            x.inc()       // nested sink
    }
    x = getNullableInt() // direct redefinition
                         //   after the nested sink
}

fun nestedSinkBad02() {
    var x: Int? = 42      // definition
    run {
        x = null          // nested redefinition
    }
    run {
        if (x != null)    // smart cast source
            x.inc()       // nested sink
    }
}
```

14.1.4 Loop handling

As mentioned before, a compiler may use *killDataFlow* instructions in loops to avoid slow data-flow analysis convergence. In the general case, a loop body may be evaluated zero or more times, which, combined with *killDataFlow* instructions, causes the smart cast sources from the loop body to *not* propagate to the containing scope. However, some loops, for which we can have static guarantees about how their body is evaluated, may be handled differently. For the following loop configurations, we consider their bodies to be definitely evaluated *one or more* times.

- `while (true) { ... }`
- `do { ... } while (condition)`

  Note: in the current implementation, only the exact `while (true)` form is handled as described; e.g., `while (true == true)` does not work.

  Note: one may extend the number of loop configurations, which are handled by smart casts, if the compiler implementation deems it necessary.

  Example:

  ```
  fun breakFromInfiniteLoop() {
      var a: Any? = null
  ```

```
    while (true) {
        if (a == null) return

        if (randomBoolean()) break
    }

    a // Smart cast to Any
}

fun doWhileAndSmartCasts() {
    var a: Any? = null

    do {
        if (a == null) return
    } while (randomBoolean())

    a // Smart cast to Any
}

fun doWhileAndSmartCasts2() {
    var a: Any? = null

    do {
        println(a)
    } while (a == null)

    a // Smart cast to Any
}
```

14.1.5 Bound smart casts

In some cases, it is possible to introduce smart casting *between properties* if it is known at compile-time that these properties are *bound* to each other. For instance, if a variable `a` is initialized as a copy of variable `b` and both are stable, they are guaranteed to reference the same runtime value and any assumption about `a` may be also applied to `b` and vice versa.

Example:

```
val a: Any? = ...
val b = a

if (b is Int) {
    // as a and b point to the same value,
    // a also is Int
    a.inc()
}
```

In more complex cases, however, it may not be trivial to deduce that two (or more) properties point to the same runtime object. This relation is known as *must-alias* relation between program references and it is implementation-defined in which cases a particular Kotlin compiler may safely assume this relation holds between two particular properties at a particular program point. However, it must guarantee that if two properties are considered bound, it is impossible for these properties to reference two different values at runtime.

One way of implementing bound smart casts would be to divide the space of stable program properties into disjoint *alias sets* of properties, and the analysis described above links the smart cast data flow information to sets of properties instead of single properties.

Such view could be further refined by considering special *alias sets* separately; e.g., an alias set of definitely non-null properties, which would allow the compiler to infer that `a?.b !== null` implies `a !== null` (for non-nullable `b`).

14.2 Local type inference

Local type inference in Kotlin is the process of deducing the compile-time types of expressions, lambda expression parameters and properties. As previously mentioned, type inference is a type constraint problem, and is usually solved by a type constraint solver.

In addition to the types of intermediate expressions, local type inference also performs deduction and substitution for generic type parameters of functions and types involved in every expression. You can use the Expressions part of this specification as a reference point on how the types for different expressions are constructed.

> Important: additional effects of smart casts are considered in local type inference, if applicable.

Type inference in Kotlin is bidirectional; meaning the types of expressions may be derived not only from their arguments, but from their usage as well. Note that, albeit bidirectional, this process is still local, meaning it processes one statement at a time, strictly in the order of their appearance in a scope; e.g., the type of property in statement S_1 that goes before statement S_2 cannot be inferred based on how S_1 is used in S_2.

As solving a type constraint system is not a definite process (there may be more than one valid solution for a given constraint system), type inference may create several valid solutions. In particular, one may always derive a constraint $A <: T <: B$ for every free type variable T, where types A and B are both valid solutions.

> Note: this is valid even if T is a free type variable without any explicit constraints, as every type in Kotlin has an implicit constraint

`kotlin.Nothing` <: T <: `kotlin.Any?`.

In these cases an optimal constraint system solution is picked w.r.t. local type inference.

> Note: for the purposes of local type inference, an optimal solution is the one which does not contain any free type variables with no explicit constraints on them.

14.3 Function signature type inference

Function signature type inference is a variant of local type inference, which is performed for function declarations, lambda literals and anonymous function declarations.

14.3.1 Named and anonymous function declarations

As described here, a named function declaration body may come in two forms: an expression body (a single expression) or a control structure body. For the latter case, an expected return type must be provided or is assumed to be `kotlin.Unit` and no special kind of type inference is needed. For the former case, an expected return type may be provided or can be inferred using local type inference from the expression body. If the expected return type is provided, it is used as an expected constraint on the result type of the expression body.

Example:

```
fun <T> foo(): T { ... }
fun bar(): Int = foo() // an expected constraint T' <: Int
// allows the result of `foo` to be inferred automatically.
```

14.3.2 Statements with lambda literals

Complex statements involving one or more lambda literals introduce an additional level of complexity to type inference and overload resolution mechanisms. As mentioned in the overload resolution section, the overload resolution of callables involved in such statements is performed regardless of the contents of the lambda expressions and before any processing of their bodies is performed (including local type inference).

For a complex statement S involving (potentially overloaded) callables $C_1, \ldots, C_N$ and lambda literals $L_1, \ldots, L_M$, excluding the bodies of these literals, they are processed as follows.

1. An empty type constraint system Q is created;
2. The overload resolution, if possible, picks candidates for $C_1, \ldots, C_N$ according to the overload resolution rules;

3. For each lambda literal with unspecified number of parameters, we decide whether it has zero or one parameter based on the form of the callables and/or the expected type of the lambda literal. If there is no way to determine the number of parameters, it is assumed to be zero. If the number of parameters is determined to be one, the phantom parameter `it` is proceeded in further steps as if it was a named lambda parameter;

   > Important: the presence or absence of the phantom parameter `it` in the lambda body does not influence this process in any way.

4. For each lambda body $L_1, \ldots, L_N$, the expected constraints on the lambda arguments and/or lambda result type from the selected overload candidates (if any) are added to Q, and the overload resolution for all statements in these bodies is performed w.r.t. updated type constraint system Q. This may result in performing steps 1-3 in a recursive *top-down* fashion for nested lambda literals;

   > Important: in some cases overload resolution may fail to pick a candidate, e.g., because the expected constraints are incomplete, causing the constraint system to be unsound. If this happens, it is implementation-defined whether the compiler continues the top-down analysis or stops abruptly.

5. When the top-down analysis is done and the overload candidates are fixed, local type inference is performed on each lambda body and each statement *bottom-up*, from the most inner lambda literals to the outermost ones, processing one lambda literal at a time, with the following additions.

   - When inferring type of the return value (the last expression of a lambda body and/or the subjects for return expressions referring to this lambda literal), the additional constraints introduced on the result type of this lambda literal are added to Q;
   - If inference with these constraints fails, but the result type is a subtype of `kotlin.Unit`, the inference is repeated without the additional constraints on the return value;
   - The type of each lambda literal is considered to be the functional type $\mathrm{FT}(P_1, \ldots, P_S) \to R$, where $P_1, \ldots, P_S$ are the types of its parameters inferred from external constraints or specified in the lambda literal itself and R is the inferred type of its return value in the presence of external constraints.

The external constraints on lambda parameters, return value and body may come from the following sources:

- The (possibly overloaded) callable which uses the lambda literal as an argument;

  > Note: as overload resolution is performed before any lambda literal inference takes place, this candidate is always known

before external constraints are needed;

- The expected type of the declaration which uses the lambda literal as its body or initializer.

  Examples:

```
fun <T> foo(): T { ... }
fun <R> run(body: () -> R): R { ... }

fun bar() {
    val x = run {
        run {
            run {
                foo<Int>() // last expression inferred to be of type Int
            } // this lambda is inferred to be of type () -> Int
        } // this lambda is inferred to be of type () -> Int
    } // this lambda is inferred to be of type () -> Int
    // x is inferred to be of type Int

    val y: Double = run { // this lambda has an external constraint R' <: Double
        run { // this lambda has an external constraint R'' <: Double
            foo() // this call has an external constraint T' <: Double
                  // allowing to infer T to be Double in foo
        }
    }
}
```

14.4 Bare type argument inference

Bare type argument inference is a special kind of type inference where, given a type T and a constructor TC, the type arguments $A_0, A_1 \dots A_N$ are inferred such that $TC[A_0, A_1 \dots A_N] <: T$. It is used together with *bare types* syntax sugar that can be employed in type checking and casting operators. The process is performed as follows.

First, let's consider the simple case of T being non-nullable, non-intersection type. Then, a simple type constraint system is constructed by introducing type variables for $A_0, A_1 \dots A_N$ and then solving the constraint $TC[A_0, A_1 \dots A_N] <: T$.

If T is an intersection type, the same process is performed for every member of the intersection type individually and then the resulting type argument values for each parameter A_K are merged using the following principle:

- If all values for a particular parameters are star-projections, the result is a star-projection;
- If some of the values are not star-projections and are strictly equal to each other, the result is one of their values;

- Else, the result is a star-projection.

If T is a nullable type $U?$, the steps given above are performed for its non-nullable counterpart type U.

14.5 Builder-style type inference

> Note: before Kotlin 1.7, builder-style type inference required using the `@BuilderInference` annotation on lambda parameters. Currently, for simple cases when there is a single lambda parameter which requires builder-style inference, this annotation may be omitted.

When working with DSLs that have generic builder functions, one may want to infer the generic builder type parameters using the information from the builder's lambda body. Kotlin supports special kind of type inference called **builder-style type inference** to allow this in some cases.

In order to allow builder-style inference for a generic builder function and its type parameter `P`, it should satisfy the following requirements:

- It has a lambda parameter of function type with receiver, with receiver type `T`
- The receiver type `T` uses type parameter `P` in its type arguments
- The receiver type `T` can be used as receiver for callables which can provide information about `P` via their use

> Note: using the type parameter `P` *directly* as the receiver type `T` (e.g., `fun <Q /* P */> myBuilder(builder: Q /* T */.() -> Unit)`) is not yet supported.

In essence, the builder-style inference allows the type of the lambda parameter receiver to be inferred from its usage in the lambda body. This is performed only if the standard type inference cannot infer the required types, meaning one could provide additional type information to help with the inference, e.g., via explicit type arguments, and avoid the need for builder-style inference.

If the builder-style inference is needed, for a call to an eligible function with a lambda parameter, the inference is performed as described above, but the type arguments of the lambda parameter receiver are viewed as *postponed* type variables till the body of the lambda expression is proceeded.

> Note: during the builder-style inference process, a postponed type variable is not required to be inferred to a concrete type.

After the inference of statements inside the lambda is complete, these postponed type variables are inferred using an additional type inference step, which takes the resulting type constraint system and tries to find the instantiation of the postponed type variables to concrete types.

If the system cannot be solved, it is a compile-time error.

Builder-style inference has the following important restrictions.

- Any attempt to use an expression with type which is a postponed type variable is a compile-time error.
- If a call needs builder-style inference for more than one lambda parameter, they all should be marked with `@BuilderInference` annotation. Otherwise, it is a compile-time error.

  Note: notable examples of builder-style inference-enabled functions are `kotlin.sequence` and `kotlin.iterator`. See standard library documentation for details.

  Example:

```
fun <K, V> buildMap(action: MutableMap<K, V>.() -> Unit): Map<K, V> { ... }

interface Map<K, out V> : Map<K, V> { ... }
interface MutableMap<K, V> : Map<K, V> {
    fun put(key: K, value: V): V?
    fun putAll(from: Map<out K, V>): Unit
}

fun addEntryToMap(baseMap: Map<String, Number>,
                  additionalEntry: Pair<String, Int>?) {
   val myMap = buildMap/* <?, ?> */ { // cannot infer type arguments
                                      // needs builder-style inference
       putAll(baseMap)
       // provides information about String <: K, Number <: V

       if (additionalEntry != null) {
           put(additionalEntry.first, additionalEntry.second)
           // provides information about String <: K, Int <: V
       }
   }
   // solves to String =:= K, Number =:= V
   // ...
}
```