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.

> Note: type inference is a [type constraint][Kotlin type constraints] problem, and is usually solved by a type constraint solver.

TODO(write about when type inference works and when it does not)

Smart casts

Kotlin introduces a limited form of flow-dependent typing called *smart casting*. Flow-dependent 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.

Smart casts are dependent on two main things: *smart cast sources* and *smart cast sink stability*.

Smart cast sources

There are two kinds of smart cast sources: *non-nullability conditions* and *type conditions*. Non-nullability conditions specify that some value is not nullable, i.e., its value is guaranteed to not be `null`. Type conditions specify that some value's runtime type conforms to a constraint $RT <: T$, where T is the assumed type and RT is the runtime type. A smart cast source may be *negated*, meaning it reverses its interpretation.

> Note: non-nullability conditions may be viewed as a special case of type conditions with assumed type `kotlin.Any`.

> Note: we may use the terms "negated non-nullability condition" and "nullability condition" interchangeably.

These sources influence the compile-time type of a value in some expression (called *smart cast sink*) only if the sink is *stable* and if the source dominates the sink. The actual compile-time type of a smart casted value for most purposes (including, but not limited to, function overloading and type inference of other values) is as follows.

- If the smart cast source is a non-nullability condition, the type is the [intersection][Type intersection] of the type it had before (including the results of smart casting performed for other conditions) and type `kotlin.Any`;
- If the smart cast source is a negated non-nullability condition, the type is the [intersection][Type intersection] of the type it had before (including the results of smart casting performed for other conditions) and type `kotlin.Nothing?`;
- If the smart cast source is a type condition, the type is the [intersection][Type intersection] of the type it had before (including the results of smart casting performed for other conditions) and the assumed type of the condition.
- If the smart cast source is a negated type condition, the type does not change.

  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 cast sources are introduced by:

- Conditional expressions (`if` and `when`);
- Elvis operator (operator `?:`);
- Safe navigation operator (operator `?.`);

- Logical conjunction expressions (operator `&&`);
- Logical disjunction expressions (operator `||`);
- Not-null assertion expressions (operator `!!`);
- Direct casting expression (operator `as`);
- Direct 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.

Nullability and type conditions are derived in the following way.

- `x is T` where x is an applicable expression implies a type condition for x with assumed type T;
- `x !is T` where x is an applicable expression implies a negated type condition for x with assumed type T;
- `x != null` or `null != x` where x is an applicable expression implies a non-nullability condition for x;
- `x == null` or `null == x` where x is an applicable expression implies a nullability condition for x;
- `!x` implies all the conditions implied by x, but in negated form;
- `x && y` implies the union of all non-negated conditions implied by x and y and the intersection of all negated conditions implied by x and y;
- `x || y` implies the union of all negated conditions implied by x and y and the intersection of all non-negated conditions implied by x and y;
- `x === y` or `y === x` where x is an applicable expression and y is a known non-nullable value (that is, has a non-nullable compile-time type) implies the non-nullability condition for x;
- `x === y` or `y === x` where x is an applicable expression and y is known to be null (that is, has `Nothing?` type) implies the nullability condition for x;
- `x == y` or `y == x` where x is an applicable expression and y is a known non-nullable value (that is, has a non-nullable compile-time type) implies the non-nullability condition for x, but only if the corresponding [`equals` implementation][Value equality expressions] is known to be equivalent to [reference equality check][Reference equality expressions].
- `x == y` or `y == x` where x is an applicable expression and y is known to be null (that is, has `Nothing?` type) implies the nullability condition for x, but only if the corresponding [`equals` implementation][Value equality expressions] is known to be equivalent to [reference equality check][Reference equality expressions].

TODO(x != Nothing? / x !== Nothing?)

  Note: for example, generated `equals` implementation for [data classes][Data class declaration] is considered to be equivalent to

> reference equality check.

TODO(A complete list of when `equals` is OK?)

Additionally, any type condition with assumed *non-null* type also creates a non-nullability condition for its value. This property is used in bound smart casts.

Smart cast sink stability

A smart cast sink is *stable* for smart casting if its value cannot be changed from the smart cast source to itself; this guarantees the smart cast conditions still hold at the sink. This is one of the necessary conditions for smart cast to be applicable for a given source-sink pair.

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

- concurrent writes;
- separate module compilation;
- custom getters;
- delegation.

> Note: despite what it may seem at first sight, sink stability is *very* complicated for local variables.

The following smart cast sinks are considered stable.

1. Immutable local or classifier-scope properties without delegation or custom getters;
2. Immutable properties of stable properties without delegation or custom getters;
3. 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 from the smart cast source to the smart cast sink.

Effectively immutable smart cast sinks

We will call redefinition of P ***direct*** redefinition, if it happens in the same declaration scope as the definition of P. If P 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 for a given pair of smart cast source SO and smart cast sink SI, if the following properties hold.

- There are no redefinitions of P on any path between SO and SI
- If SI is a direct sink, there must be no nested redefinitions on any path between D and SI
- If SI is a nested sink, then
  - there must be no nested redefinitions of P
  - all direct redefinitions of P must precede SI

  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 sunk
    }

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

Source-sink domination

A smart cast source SO dominates a smart cast sink SI, if SO is a control-flow dominator of SI. This is one of the necessary conditions for smart cast to be applicable for a given source-sink pair.

In the most basic case, smart cast conditions propagate as-is from sources to sinks. However, as a number of expressions have additional semantics, which may influence smart cast conditions, in some cases these conditions are modified along the sink-source chain. This means the following for different smart cast sources.

- Conditional expressions (`if` and `when`):
  - Smart cast conditions derived from expression condition are active inside the true branch scope;
  - Smart cast conditions derived from *negated* expression condition are active inside the false branch scope;
  - If a branch is statically known to be definitely evaluated, that branch's condition is also propagated over to its containing scope after the conditional expression;
- Elvis operator (operator `?:`): if the right-hand side of elvis operator is unreachable, a nullability condition for the left-hand side expression (if applicable) is introduced for the rest of the containing scope;
- Safe navigation operator (operator `?.`) TODO()
- Logical conjunction expressions (operator `&&`): all conditions derived from the left-hand expression are applied to the right-hand expression;
- Logical disjunction expressions (operator `||`): all conditions derived from the left-hand expression are applied *negated* to the right-hand expression;
- Not-null assertion expressions (operator `!!`): a nullability condition for the left-hand side expression (if applicable) is introduced for the rest of the containing scope;
- Unsafe cast expression (operator `as`): a type condition for the left-hand side expression (if applicable) is introduced for the rest of the containing scope; the assumed type is the same as the right-hand side type of the cast expression;
- Direct assignment: if both sides of the assignment are applicable expressions, all the conditions currently applying to the right-hand side are also applied to the left-hand side of the assignment for the rest of the containing scope.

The necessity of source-sink domination also mean that smart cast sources from the loop bodies and conditions are **not** propagated to the containing scope, as the loop body may be evaluated zero or more times, and the corresponding conditions may or may not be true. However, some loop configurations, for which we can have static guarantees about source-sink domination w.r.t. the containing scope, are handled differently.

- do-while loops (as their body is evaluated at least once) propagate the following to the rest of the containing scope:
  - smart cast sources from the loop body, which definitely dominate their sinks
  - smart cast conditions arising from the *negated* loop condition, if the loop body does not contain any `break` expressions
- `while (true)` loops propagate the following to the rest of the containing scope:

  - smart cast sources from the loop body, which definitely dominate their sinks

Note: in the second case, 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 casting, if the implementation can statically guarantee the source-sink domination.

Example:

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

    while (true) {
        if (a == null) continue

        if (randomBoolean()) break
    }

    a // Smart cast to Any
}

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

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

    a // Smart cast to Any
}

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

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

    a // Smart cast to Any
}
```

Bound smart casts

Smart casting propagates information forward on the control flow, as by the source-sink domination. However, in some cases it is beneficial to propagate information *backwards*, to reduce boilerplate code. Kotlin supports this feature by bound smart casts.

Bound smart casts apply in the following case. Assume we have two inter-dependent or bound values a and b. Bound smart casts allow to apply smart cast sources for a to b or vice versa, if both values are stable.

Kotlin supports the following bound smart casts (BSC).

- Non-nullability-by-equality BSC. If two values are known to be equal, non-nullability conditions for one are applied to the other.
- Non-nullability-by-safe-call BSC. For a safe-call property `o?.p` of a non-null type T, non-nullability conditions for `o?.p` are applied to `o`.

Two values a and b are considered equals in the following cases.

- there is a known equality or referential-equality condition between a and b
- a is definitely assigned b
  - however, in this case bound smart casts are applied only to b
  - TODO(Why?)

TODO(Do we need additional condition kinds?)

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 mentioned above, type inference is a [type constraint][Kotlin type constraints] 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][Expressions] part of this specification as a reference point on how the types for different expressions are constructed.

However, there are some additional clarifications on how these types are constructed. First, the additional effects of smart casting are considered in local type inference, if applicable. Second, there are several special cases.

- If a type T is described as the least upper bound of types A and B, it is represented as a pair of constraints $A <: T$ and $B <: T$;

- TODO(are there other cases?)

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 constraint solving]), type inference in general may have several valid solutions. In particular, one may always derive a system $A <: T <: B$ for every type variable T, where A and B are both valid solution types. One of these types is always the solution in Kotlin (although from the constraint viewpoint, there are usually more solutions available), but choosing between them is done according to the following rules:

- TODO(what are the rules?)

  Note: this is valid even if T is a variable without any explicit constraints, as every type in Kotlin has an implicit constraint `kotlin.Nothing` $<: T <:$ `kotlin.Any`?.

TODO

- Type approximation for public usage
- Ordering of lambdas (and ordering of overloading vs type inference in general)