Kotlin language specification

Version 1.9-rfc+0.1

Marat Akhin Mikhail Belyaev

Chapter 11

Overload resolution

Glossary

`type(e)`
: Type of expression `e`

Introduction

Kotlin supports *overloading* for callables and properties, that is, the ability for several callables (functions or function-like properties) or properties with the same name to coexist in the same scope, with the compiler picking the most suitable one when such entity is referenced. This section describes *overload resolution process* in detail.

> Note: most of this section explains the overload resolution process for callables, as the overload resolution process for properties uses the same framework. Important differences w.r.t. properties are covered in the corresponding section.

Unlike many object-oriented languages, Kotlin does not have only regular class methods, but also top-level functions, local functions, extension functions and function-like values, which complicate the overload resolution process quite a bit. Additionally, Kotlin has infix functions, operator and property overloading, which add their own specifics to this process.

11.1 Basics

11.1.1 Receivers

Every function or property that is defined as a method or an extension has one

or more special parameters called *receiver* parameters. When calling such a callable using navigation operators (`.` or `?.`) the left hand side value is called an *explicit receiver* of this particular call. In addition to the explicit receiver, each call may indirectly access zero or more *implicit receivers*.

Implicit receivers are available in a syntactic scope according to the following rules:

- Any receiver available in a scope is available in its downwards-linked scopes;
- In a classifier declaration scope (including object and companion object declarations), the declared object is available as implicit `this`;
- In a classifier declaration scope (including object and companion object declarations), the static callables of the declared object are available on a phantom static implicit `this`;
- If a function or a property is an extension, `this` parameter of the extension is also available inside the extension declaration;
- If a lambda expression has an extension function type, `this` argument of the lambda expression is also available inside the lambda expression declaration.

  Important: a phantom static implicit `this` is a special receiver, which is included in the receiver chain for the purposes of handling static functions from enum classes. It may also be used on platforms to handle their static-like entities, e.g., static methods on JVM platform.

The available receivers are prioritized in the following way:

- Receivers provided in the most inner scope have higher priority as ordered w.r.t. link relation;
- The implicit `this` receiver has higher priority than phantom static implicit `this`;
- The phantom static implicit `this` receiver has higher priority than the current class companion object receiver;
- Current class companion object receiver has higher priority than any of the superclass companion objects;
- Superclass companion object receivers are prioritized according to the inheritance order.

  Important: these rules mean implicit receivers are always totally ordered w.r.t. their priority, as no two implicit receivers can have the same priority.

  Important: DSL-specific annotations (marked with `kotlin.DslMarker` annotation) change the availability of implicit receivers in the following way: for all types marked with a particular DSL-specific annotation, only the highest priority implicit receiver is available in a given scope.

The implicit receiver having the highest priority is also called the *default implicit receiver*. The default implicit receiver is available in a scope as `this`. Other

available receivers may be accessed using labeled this-expressions.

If an implicit receiver is available in a given scope, it may be used to call callables implicitly in that scope without using the navigation operator.

For extension callables, the receiver used as the extension receiver parameter is called *extension receiver*, while the implicit receiver associated with the declaration scope the extension is declared in is called *dispatch receiver*. For a particular callable invocation, any or both receivers may be involved, but, if an extension receiver is involved, the dispatch receiver must be implicit.

> Note: by definition, local extension callables do not have a dispatch receiver, as they are declared in a statement scope.
>
> Note: there may be situations in which *the same implicit receiver* is used as both the dispatch receiver and the extension receiver for a particular callable invocation, for example:

```
interface Y

class X : Y {
    fun Y.foo() {} // `foo` is an extension for Y,
                   //   needs extension receiver to be called

    fun bar() {
        foo() // `this` reference is both
              //   the extension and the dispatch receiver
    }
}

fun <T> mk(): T = TODO()

fun main() {
    val x: X = mk()
    val y: Y = mk()

    // y.foo()
    // Error, as there is no implicit receiver
    //   of type X available

    with (x) {
        y.foo() // OK!
    }
}
```

11.1.2 The forms of call-expression

Any function in Kotlin may be called in several different ways:

- A fully-qualified call without receiver: `package.foo()`;
- A call with an explicit receiver: `a.foo()`;
- An infix function call: `a foo b`;
- An overloaded operator call: `a + b`;
- A call without an explicit receiver: `foo()`.

Although syntactically similar, there is a difference between the first two kinds of calls: in the first case, `package` is a name of a Kotlin package, while in the second case `a` is a value or a type.

For each of these cases, a compiler should first pick a number of *overload candidates*, which form a set of possibly intended callables (*overload candidate set*, OCS), and then *choose the most specific function* to call based on the types of the function and the call arguments.

> Important: the overload candidates are picked **before** the most specific function is chosen.

11.1.3 Callables and `invoke` convention

A *callable* X for the purpose of this section is one of the following:

- Function-like callables:
  - A function named X at its declaration site;
  - A constructor of a type named X at its declaration site;
  - Any of the above named Y at its declaration site, but imported into the current scope using a renaming import as X.
- Property-like callables with an operator function `invoke` available as a member or an extension in the current scope:
  - A property named X at its declaration site;
  - An object or a companion object named X at its declaration site;
  - A companion object of a classifier type named X at its declaration site;
  - An enum entry named X at its declaration site;
  - Any of the above named Y at its declaration site, but imported into the current scope using a renaming import as X.

For property-like callables, a call $X(Y_0, \ldots, Y_N)$ is an overloadable operator which is expanded to X.invoke$(Y_0, \ldots, Y_N)$. The call may contain type parameters, named parameters, variable argument parameter expansion and trailing lambda parameters, all of which are forwarded as-is to the corresponding `invoke` function.

The set of implicit receivers itself (denoted by `this` expression) may also be used as a property-like callable using `this` as the left-hand side of the call expression. As with normal property-like callables, `this@A`$(Y_0, \ldots, Y_N)$ is an overloadable operator which is expanded to `this@A.invoke`$(Y_0, \ldots, Y_N)$.

A *member callable* is one of the following:

- a member function-like callable (including constructors);

- a member property-like callable with a member operator `invoke`.

An *extension callable* is one of the following:

- an extension function-like callable;
- a member property-like callable with an extension operator `invoke`;
- an extension property-like callable with a member operator `invoke`;
- an extension property-like callable with an extension operator `invoke`.

  Informally: the mnemonic rule to remember this order is "functions before properties, members before extensions".

A *local callable* is any callable which is declared in a statement scope.

11.1.4 c-level partition

When calculating overload candidate sets, member callables produce the following sets, considered separately, ordered by higher priority first:

- Member function-like callables;
- Member property-like callables.

Extension callables produce the following sets, considered separately, ordered by higher priority first:

- Extension function-like callables;
- Member property-like callables with extension `invoke`;
- Extension property-like callables with member `invoke`;
- Extension property-like callables with extension `invoke`.

Let us define this partition of callables to overload candidate sets as *c-level partition* (callable-level partition). As this partition is the most fine-grained of all other steps of partitioning resolution candidates into sets, it is always performed **last**, after all other applicable steps.

11.2 Building the overload candidate set

11.2.1 Fully-qualified call

If a call is fully-qualified (that is, it contains a complete package path), then the overload candidate set S simply contains all the top-level callables with the specified name in the specified package. As a package name can never clash with any other declared entity, after performing c-level partition on S, the resulting sets are the only ones available for further processing.

Example:

```
package a.b.c

fun foo(a: Int) {}
```

```
fun foo(a: Double) {}
fun foo(a: List<Char>) {}
val foo = {}
. . .
a.b.c.foo()
```

Here the resulting overload candidate set contains all the callables named `foo` from the package `a.b.c`.

Important: a fully-qualified callable name has the form `P.n()`, where `n` is a simple callable name and `P` is a complete package path referencing an existing package.

11.2.2 Call with an explicit receiver

If a call is done via a navigation operator (`.` or `?.`), but is not a fully-qualified call, then the left hand side value of the call is the explicit receiver of this call.

A call of callable `f` with an explicit receiver `e` is correct if at least one of the following holds:

1. `f` is an accessible member callable of the classifier type `type(e)` or any of its supertypes;
2. `f` is an accessible extension callable of the classifier type `type(e)` or any of its supertypes, including top-level, local and imported extensions.
3. `f` is an accessible static member callable of the classifier type `e`.

   Important: callables for case 2 include not only regular extension callables, but also extension callables from any of the available implicit receivers. For example, if class `P` contains a member extension function `f` for another class `T` and an object of class `P` is available as an implicit receiver, extension function `f` may be used for such call if `T` conforms to the type `type(e)`.

If a call is correct, for a callable `f` with an explicit receiver `e` of type `T` the following sets are analyzed (**in the given order**):

1. Non-extension member callables named `f` of type `T`;
2. Extension callables named `f`, whose receiver type `U` conforms to type `T`, in the current scope and its upwards-linked scopes, ordered by the size of the scope (smallest first), excluding the package scope;
   - First, we assume there is **no implicit receiver** available for the dispatch receiver of `f` (i.e., we analyze *local* extension callables only);
   - Second, we consider each implicit receiver available for the dispatch receiver of `f` in the order of the implicit receiver priority;
3. Explicitly imported extension callables named `f`, whose receiver type `U` conforms to type `T`;
4. Extension callables named `f`, whose receiver type `U` conforms to type `T`, declared in the package scope;

5. Star-imported extension callables named `f`, whose receiver type `U` conforms to type `T`;
6. Implicitly imported extension callables named `f` (either from the Kotlin standard library or platform-specific ones), whose receiver type `U` conforms to type `T`.

> Note: here type `U` conforms to type `T`, if $T <: U$.

There is a important special case here, however, as a callable may be a property-like callable with an operator function `invoke`, and these may belong to different sets (e.g., the property itself may be star-imported, while the `invoke` operator on it is a local extension). In this situation, such callable belongs to the **lowest priority** set of its parts (e.g., for the above case, priority 5 set).

> Example: when trying to resolve between an explicitly imported extension property (priority 3) with a member `invoke` (priority 1) and a local property (priority 2) with a star-imported extension `invoke` (priority 5), the first one wins (`max(3, 1) < max(2, 5)`).

When analyzing these sets, the **first** set which contains **any** applicable callable is picked for c-level partition, which gives us the resulting overload candidate set.

> Important: this means, among other things, that if the set constructed on step Y contains the overall most suitable candidate function, but the set constructed on step $X < Y$ is not empty, the callables from set X will be picked despite them being less suitable overload candidates.

After we have fixed the overload candidate set, we search this set for the most specific callable.

Call with an explicit type receiver

A call with an explicit receiver may be performed not only on a value receiver, but also on a *type* receiver.

> Note: type receivers can appear when working with enum classes or interoperating with platform-dependent code.

They mostly follow the same rules as calls with an explicit value receiver. However, for a callable `f` with an explicit type receiver `T` the following sets are analyzed (**in the given order**):

1. Static member callables named `f` of type `T`;
2. Static member callables named `f` of type `T` declared implicitly;
3. The overload candidate sets for call `T.f()`, where `T` is a companion object of type `T`.

Call with an explicit super-form receiver

A call with an explicit receiver may be performed not only on a value receiver, but also on a super-form receiver.

They mostly follow the same rules as calls with an explicit value receiver. However, there are some differences which we outline below.

For a callable `f` with an explicit basic super-form receiver `super` in a classifier declaration with supertypes `A1`, `A2`, `...` , `AN` the following sets are considered for **non-emptiness**:

1. Non-extension member callables named `f` of type `A1`;
2. Non-extension member callables named `f` of type `A2`;
3. ...;

n. Non-extension member callables named `f` of type `AN`.

If at least two of these sets are non-empty, this is a compile-time error. Otherwise, the non-empty set (if any) is analyzed as usual.

For a callable `f` with an explicit extended super-form receiver `super<A>` the following sets are analyzed (**in the given order**):

1. Non-extension member callables named `f` of type `A`.

Additionally, in either case, abstract callables are not considered valid candidates for the overload resolution process.

11.2.3 Infix function call

Infix function calls are a special case of function calls with explicit receiver in the left hand side position, i.e., `a foo b` may be an infix form of `a.foo(b)`.

However, there is an important difference: during the overload candidate set construction the only callables considered for inclusion are the ones with the `infix` modifier. This means we consider only function-like callables with `infix` modifier and property-like callables with an `infix` operator function `invoke`. All other callables are not considered for inclusion. Aside from this difference, candidates are selected using the same rules as for normal calls with explicit receiver.

> Important: this filtering is done **before** we perform selection of the overload candidate set w.r.t. rules for calls with explicit receiver.

Different platform implementations may extend the set of functions considered as infix functions for the overload candidate set.

11.2.4 Operator call

According to the operator overloading section, some operator expressions in Kotlin can be overloaded using definition-by-convention via specifically-named functions. This makes operator expressions semantically equivalent to function calls with explicit receiver, where the receiver expression is selected based on the operator used.

However, there is an important difference: during the overload candidate set construction the only functions considered for inclusion are the ones with the `operator` modifier. All other functions (and any properties) are not considered for inclusion. Aside from this difference, candidates are selected using the same rules as for normal calls with explicit receiver.

> Important: this filtering is done **before** we perform selection of the overload candidate set w.r.t. rules for calls with explicit receiver.
>
> Note: this also means that all the properties available through the `invoke` convention are non-eligible for operator calls, as there is no way of specifying the `operator` modifier for them; even though the `invoke` callable is required to always have such modifier. As `invoke` convention itself is an operator call, it is impossible to use more than one `invoke` convention in a single call.

Different platform implementations may extend the set of functions considered as operator functions for the overload candidate set.

> Note: these rules are valid not only for dedicated operator expressions, but also for other operator-based defined-by-convention calls, e.g., `for`-loop iteration conventions, operator-form assignments or property delegation.

11.2.5 Call without an explicit receiver

A call which is performed with a simple path is a call **without** an explicit receiver. As such, it may either have one or more implicit receivers or reference a top-level function.

> Note: this case does not include calls using the `invoke` operator function where the left-land side of the call is not an identifier, but some other kind of expression (as this is not a simple path). These cases are handled the same way as operator calls and need no further special treatment.
>
> Example:

```
fun foo(a: Foo, b: Bar) {
    (a + b)(42)
    // Such a call is handled as if it is
    //   (a + b).invoke(42)
}
```

As with calls with explicit receiver, we first pick an overload candidate set and then search this set for the most specific function to match the call.

For an identifier named `f` the following sets are analyzed (**in the given order**):

1. Local non-extension callables named `f` in the current scope and its upwards-linked scopes, ordered by the size of the scope (smallest first), excluding

the package scope;
2. The overload candidate sets for each pair of implicit receivers `e` and `d` available in the current scope, calculated as if `e` is the explicit receiver, in order of the receiver priority;
3. Top-level non-extension functions named `f`, in the order of:
   a. Callables explicitly imported into the current file;
   b. Callables declared in the same package;
   c. Callables star-imported into the current file;
   d. Implicitly imported callables (either from the Kotlin standard library or platform-specific ones).

Similarly to how it works for calls with explicit receiver, a property-like callable with an `invoke function` belongs to the **lowest priority** set of its parts.

When analyzing these sets, the **first** set which contains **any** callable with the corresponding name and conforming types is picked for c-level partition, which gives us the resulting overload candidate set.

After we have fixed the overload candidate set, we search this set for the most specific callable.

11.2.6 Call with named parameters

Calls in Kotlin may use named parameters in call expressions, e.g., `f(a = 2)`, where `a` is a parameter specified in the declaration of `f`. Such calls are treated the same way as normal calls, but the overload resolution sets are filtered to only contain callables which have matching formal parameter names for all named parameters from the call.

> Important: this filtering is done **before** we perform selection of the overload candidate set w.r.t. rules for the respective type of call.
>
> Note: for properties called via `invoke` convention, the named parameters must be present in the declaration of the `invoke` operator function.

Unlike positional arguments, named arguments are matched by name directly to their respective formal parameters; this matching is performed separately for each function candidate.

While the *number of defaults* does affect resolution process, the fact that some argument was or was not mapped as a named argument does not affect this process in any way.

11.2.7 Call with trailing lambda expressions

A call expression may have a single lambda expression placed outside of the argument list or even completely replacing it (see this section for further details). This has no effect on the overload resolution process, aside from the argument

reordering which may happen because of variable length parameters or parameters with defaults.

> Example: this means that calls `f(1, 2) { g() }` and `f(1, 2, body = { g() })` are completely equivalent w.r.t. the overload resolution, assuming `body` is the name of the last formal parameter of `f`.

11.2.8 Call with specified type parameters

A call expression may have a type argument list explicitly specified before the argument list (see this section for further details). Such calls are treated the same way as normal calls, but the overload resolution sets are filtered to only contain callables which contain exactly the same number of formal type parameters at declaration site. In case of a property-like callable with `invoke`, type parameters must be present at the `invoke` operator function declaration instead.

> Important: this filtering is done **before** we perform selection of the overload candidate set w.r.t. rules for the respective type of call.

11.3 Determining function applicability for a specific call

11.3.1 Rationale

A function is *applicable* for a specific call if and only if the function parameters may be assigned the arguments values specified at the call site and all type constraints of the function type parameters hold w.r.t. supplied or inferred type arguments.

11.3.2 Description

Determining function applicability for a specific call is a type constraint problem.

First, for every non-lambda argument of the function called, type inference is performed. Lambda arguments are excluded, as their type inference needs the results of overload resolution to finish.

Second, the following constraint system is built:

- For every non-lambda argument inferred to have type T_i, corresponding to the function parameter of type U_j, a constraint $T_i <: U_j$ is constructed;
- All declaration-site type constraints for the function are also added to the constraint system;
- For every lambda argument with the number of lambda arguments known to be K, corresponding to the function parameter of type U_m, a special constraint of the form $(\mathrm{FT}(L_1, \ldots, L_K) \to R \,\&\, \mathrm{FTR}(\mathrm{RT}, L_1, \ldots, L_n) \to R) <: U_m$ is added to the constraint system, where $R, \mathrm{RT}, L_1, \ldots, L_K$ are fresh type variables;

- For each lambda argument with an unknown number of lambda arguments (that is, being equal to 0 or 1), corresponding to the function parameter of type U_n, a special constraint of the form $(\text{FT}() \to R \,\&\, \text{FT}(L) \to R \,\&\, \text{FTR}(\text{RT}) \to R \,\&\, \text{FTR}(\text{RT}, L) \to R) \;<:\; U_m$ is added to the constraint system, where R, RT, L are fresh type variables;

If this constraint system is sound, the function is applicable for the call. Only applicable functions are considered for the next step: choosing the most specific candidate from the overload candidate set.

Receiver parameters are handled in the same way as other parameters in this mechanism, with one important exception: any receiver of type `kotlin.Nothing` is deemed not applicable for any member callables, regardless of other parameters. This is due to the fact that, as `kotlin.Nothing` is the subtype of any other type in Kotlin type system, it would have allowed **all** member callables of **all** available types to participate in the overload resolution, which is theoretically possible, but very resource-consuming and does not make much sense from the practical point of view. Extension callables are still available, because they are limited to the declarations available or imported in the current scope.

> Note: although it is impossible to create a value of type `kotlin.Nothing` directly, there may be situations where performing overload resolution on such value is necessary; for example, it may occur when doing safe navigation on values of type `kotlin.Nothing?`.

11.4 Choosing the most specific candidate from the overload candidate set

11.4.1 Rationale

The main rationale for choosing the most specific candidate from the overload candidate set is the following:

> The most specific callable can forward itself to any other callable from the overload candidate set, while the opposite is not true.

If there are several functions with this property, none of them are the most specific and an overload resolution ambiguity error should be reported by the compiler.

Consider the following example.

```
fun f(arg: Int, arg2: String) {}        // (1)
fun f(arg: Any?, arg2: CharSequence) {} // (2)
...
f(2, "Hello")
```

Both functions (1) and (2) are applicable for the call, but function (1) could easily call function (2) by forwarding both arguments into it, and the reverse is impossible. As a result, function (1) is more specific of the two.

```
fun f1(arg: Int, arg2: String) {
    f2(arg, arg2) // VALID: can forward both arguments
}
fun f2(arg: Any?, arg2: CharSequence) {
    f1(arg, arg2) // INVALID: function f1 is not applicable
}
```

The rest of this section will describe how the Kotlin compiler checks for this property in more detail.

11.4.2 Algorithm of MSC selection

When an overload resolution set S is selected and it contains more than one callable, we need to choose the most specific candidate from these callables. The selection process uses the type constraint facilities of Kotlin, in a way similar to the process of determining function applicability.

For every two distinct members of the candidate set F_1 and F_2, the following constraint system is constructed and solved:

- For every non-default argument of the call and their corresponding declaration-site parameter types $X_1, \ldots, X_N$ of F_1 and $Y_1, \ldots, Y_N$ of F_2, a type constraint $X_K <: Y_K$ is built **unless both X_K and Y_K are built-in integer types.** If both X_K and Y_K are built-in integer types, a type constraint $\text{Widen}(X_K) <: \text{Widen}(Y_K)$ is built instead, where Widen is the integer type widening operator. During construction of these constraints, all declaration-site type parameters $T_1, \ldots, T_M$ of F_1 are considered bound to fresh type variables $T_1^\sim, \ldots, T_M^\sim$, and all type parameters of F_2 are considered free;
- If F_1 and F_2 are extension callables, their extension receivers are also considered non-default arguments of the call, even if implicit, and the corresponding constraints are added to the constraint system as stated above. For non-extension callables, only declaration-site parameters are considered;
- All declaration-site type constraints of $X_1, \ldots, X_N$ and $Y_1, \ldots, Y_N$ are also added to the constraint system.

  Note: this constraint system checks whether F_1 can forward itself to F_2.

If the resulting constraint system is sound, it means that F_1 is equally or more applicable than F_2 as an overload candidate (aka applicability criteria). The check is then repeated with F_1 and F_2 swapped.

This check may result in one of the following outcomes:

1. Only one of the two candidates is more applicable than the other;
2. Neither of the two candidates is more applicable than the other;
3. Both F_1 and F_2 are more applicable than the other.

In case 1, the more applicable candidate of the two is found and no additional steps are needed.

In case 2, an additional step is performed.

- Any non-parameterized callable is a more specific candidate than any parameterized callable; If there are several non-parameterized candidates, further steps are limited to those candidates.

In case 3, several additional steps are performed in order.

- Any non-parameterized callable is a more specific candidate than any parameterized callable (same as case 2). If there are several non-parameterized candidates, further steps are limited to those candidates;
- For each candidate we count the number of default parameters *not* specified in the call (i.e., the number of parameters for which we use the default value). The candidate with the least number of non-specified default parameters is a more specific candidate;
- For all candidates, the candidate having any variable-argument parameters is less specific than any candidate without them.

  Note: it may seem strange to process built-in integer types in a way different from other types, but it is needed for cases when the call argument is an integer literal with an integer literal type. In this particular case, several functions with different built-in integer types for the corresponding parameter may be applicable, and the `kotlin.Int` overload is selected to be the most specific.

  Important: compiler implementations may extend these steps with additional checks, if they deem necessary to do so.

If after these additional steps there are still several candidates which are equally applicable for the call, we may attempt to use the lambda return type to refine function applicability. If there are still more than one most specific candidate afterwards, this is an **overload ambiguity** which must be reported as a compile-time error.

> Note: unlike the applicability test, the candidate comparison constraint system is **not** based on the actual call, meaning that, when comparing two candidates, only constraints visible at *declaration site* apply.

If the callables in check are properties with available `invoke`, the same process is applied in two steps:

- First, the properties are compared for applicability and the most applicable property is chosen as described above. If several properties are equally

applicable, this is an overload ambiguity as usual;
- Second, for the property selected at first step, the most applicable operator `invoke` overload is chosen.

11.4.3 Using lambda return type to refine function applicability

If the most specific candidate set C is ambiguous (has more than one callable) and contains at least one callable marked with `kotlin.OverloadResolutionByLambdaReturnType`, several additional checks and steps are performed to reduce it, by attempting to infer a single lambda return type and use it to refine function applicability.

First, we perform the following checks.

1. We *check* if the function call contains **exactly one** lambda argument A which requires type inference (which does not have an explicitly defined type).
2. For every function in C we collect parameters P_i corresponding to argument A and *check* their function types T_i to be structurally equal excluding return types (SEERT).

   Informally: SEERT checks whether function types have the exactly same input parameters.

   Examples: the following two function types are considered SEERT.

   - `(Int, String) -> Int`
   - `(Int, String) -> Double`

   The following two function types are not considered SEERT.

   - `Int.(String) -> Int`
   - `(Int, String) -> Double`

If all checks succeed, we can perform the type inference for the lambda argument A, as in all cases its parameter types are known (corollary from check 2 succeeding) and their corresponding constraints can be added to the constraint system. The constraint system solution gives us the inferred lambda return type R_{inf}, which may be used to refine function applicability, by removing overload candidates with incompatible lambda return types.

This is performed by repeating the function applicability test on the most specific candidate set C, with the additional constraint $R \equiv R_{\text{inf}}$ added for the corresponding lambda argument A. Candidates which remain applicable with this additional constraint are added to the refined set C'.

> Note: If any of the checks described above fails, we continue with the set $C' = C$.

If set C' contains more than one candidate, we attempt to prefer candidates **without** `kotlin.OverloadResolutionByLambdaReturnType` annotation. If there

are any, they are included in the resulting most specific candidate set C_{res}, with which we finish the MSC selection. Otherwise, we finish the MSC selection with the set C'.

Example:

```
@OverloadResolutionByLambdaReturnType
fun foo(cb: (Unit) -> String) = Unit // (1)

@OverloadResolutionByLambdaReturnType
fun foo(cb: (Unit) -> Int) = Unit // (2)

fun testOk01() {
    foo { 42 }
    // Both (1) and (2) are applicable
    // (2) is preferred by the lambda return type
}
```

Example:

```
@OverloadResolutionByLambdaReturnType
fun foo(cb: Unit.() -> String) = Unit // (1)

@OverloadResolutionByLambdaReturnType
fun foo(cb: (Unit) -> Int) = Unit // (2)

fun testError01() {
    val take = Unit
    // Overload ambiguity
    foo { 42 }
    // Both (1) and (2) are applicable
    // None is preferred by the lambda return type
    //   as their parameters are not SEERT
}
```

Example:

```
@OverloadResolutionByLambdaReturnType
fun foo(cb: Unit.() -> String) = Unit // (1)

@OverloadResolutionByLambdaReturnType
fun foo(cb: (Unit) -> Int) = Unit // (2)

fun testOk02() {
    val take = Unit
    foo { a -> 42 }
    // Only (2) is applicable
    //   as its lambda takes one parameter
}
```

Example:

```
@OverloadResolutionByLambdaReturnType
fun foo(cb: (Unit) -> String) = Unit // (1)

@OverloadResolutionByLambdaReturnType
fun foo(cb: (Unit) -> CharSequence) = Unit // (2)

fun testError02() {
    // Error: required String, found CharSequence
    foo { a ->
        val a: CharSequence = "42"
        a
    }
    // Both (1) and (2) are applicable
    // (1) is the only most specific candidate
    //   We do not attempt refinement by the lambda return type
}
```

11.5 Resolving property access

As properties in Kotlin can have custom getters and setters, be extension or delegated, they are also subject to overload resolution. Overload resolution for property access works similarly to how it works for callables, i.e., it consists of two steps: building the overload candidate set of applicable candidates, and then choosing the most specific candidate from the overload candidate set.

> *Important*: this section concerns *only* properties accessed using property access syntax `a.x` or just `x` *without call suffix*. If a property is accessed with a call suffix, it is treated as any other callable and is required to have a suitable `invoke` overload available, see the rest of this part for details

There are two variants of property access syntax: read-only property access and property assignment.

> Note: there is also safe navigation syntax for both assignment and read-only access, but that is expanded to non-safe navigation syntax covered by this section. Please refer to corresponding sections for details.

Read-only property access `a.x` is resolved the same way as if the property access in question was a special function call `a.x$get()` and each property `val/var x: T` was replaced with corresponding function `fun x$get(): T` having all the same extension receivers, context receivers, type parameters and scope as the original property and providing direct access to the property getter. For different flavors of property declarations and getters, refer to corresponding section. Please

note that this excludes any possibility to employ `invoke`-convention as these ephemeral functions cannot be properties themselves.

> Example: one may consider property access in class `A` to be resolved as if it has been transformed to class `AA`.

```
class A {
    val a: Int = 5 // (1)

    val Double.a: Boolean // (2)
        get() = this != 42.0

    fun test() {

        println(a) // Resolves to (1)

        with(42.0) {
            println(this@A.a) // Resolves to (1)
            println(this.a) // Resolves to (2)
            println(a) // Resolves to (2)
        }
    }
}

class AA {
    fun a$get(): Int = 5 // (1)

    fun Double.a$get(): Boolean // (2)
              = this != 42.0

    fun test() {

        println(a$get()) // Resolves to (1)

        with(42.0) {
            println(this@AA.a$get()) // Resolves to (1)
            println(this.a$get()) // Resolves to (2)
            println(a$get()) // Resolves to (2)
        }
    }
}
```

Property assignment `a.x = y` is resolved the same way as if it was replaced with a special function call `a.x$set(y)` and each property `var/val x: T` was replaced with a corresponding function `fun x$set(value: T)` having all the same extension receiver parameters, context receiver parameters, type parameters and scope as the original property and providing direct access to the property setter. For different flavors of property declarations and setters, refer to corresponding

section. Please note that, although a read-only property declaration (using the keyword `val`) does not allow for assignment or having a setter, it still takes part in overload resolution for property assignment and may still be picked up as a candidate. Such a candidate (in case it is selected as the final candidate) will result in compiler error at later stages of compilation.

> Note: informally, one may look at property assignment resolution as a sub-kind of read-only property resolution described above, first resolving the property as if it was accessed in a read-only fashion, and then using the setter. Read-only property access and property assignment syntax used in the same position **never** resolve to different property candidates
>
> Example: one may consider property access in class `B` to be resolved as if it has been transformed to class `BB`. Declaration bodies for ephemeral functions are omitted to avoid confusion

```
class B {
    var b: Int = 5 // (1)

    val Double.b: Int // (2)
        get() = this.toInt()

    fun test() {
        b = 5 // Resolves to (1)

        with(42.0) {
            // Resolves to (1)
            this@B.b = 5
            // Resolves to (2) and compiler error: cannot assign read-only property
            this.b = 5
            // Resolves to (2) and compiler error: cannot assign read-only property
            b = 5
        }
    }
}

class BB {
    fun b$get(): Int // (1, getter)
    fun b$set(value: Int) // (1, setter)

    fun Double.b$get(): Int // (2, getter)
    fun Double.b$set(value: Int) // (2, setter)

    fun test() {
        b$set(5) // Resolves to (1)

        with(42.0) {
```

```
                // Resolves to (1)
                this@B.b$set(5)
                // Resolves to (2)
                this.b$set(5)
                // Resolves to (2)
                this.b$set(5)
            }
        }
    }
```

The overload resolution for properties has the following features distinct from overload resolution for callables.

- Properties without getter or setter are assumed to have default implementations for accessors (ones which get or set its backing field);
- The overload resolution takes into account the kind of property, meaning an extension read-only property is considered to have an extension getter, an extension mutable property is considered to have an extension getter and setter, etc.;
- Object declarations and enumeration entries may be accessed using the property access syntax given that they may be resolved in the current scope.

11.6 Resolving callable references

Callable references introduce a special case of overload resolution which is somewhat similar to how regular calls are resolved, but different in several important aspects.

First, property and function references are treated equally, as both kinds of references have a type which is a subtype of a function type. Second, the type information needed to perform the resolution steps is acquired from *expected type* of the reference itself, rather than the types of arguments and/or result. The `invoke` operator convention **does not** apply to callable reference candidates. Third, and most important, is that, in the case of a call with a callable reference as a parameter, the resolution is **bidirectional**, meaning that both the callable being called and the callable being referenced are to be resolved *simultaneously*.

11.6.1 Resolving callable references not used as arguments to a call

In a simple case when the callable reference is not used as an argument to an overloaded call, its resolution is performed as follows:

- For each callable reference candidate, we perform the following steps:
  - We build its type constraints and add them to the constraint system of the expression the callable reference is used in;

  - A callable reference is deemed applicable if the constraint system is sound;
- For all applicable candidates, the resolution sets are built according to the same rules as building OCS for regular calls;
- If the highest priority set contains more than one callable, this is an overload ambiguity and should be reported as a compile-time error.
- Otherwise, the single callable in the set is chosen as the result of the resolution process.

  Note: this is different from the overload resolution for regular calls in that no most specific candidate selection process is performed inside the sets

  Important: when the callable reference resolution for `T::f` requires building overload candidate sets for both type and value receiver candidates, they are considered in the following order.

  1. Static member callables named `f` of type `T`;
  2. The overload candidate sets for call `t::f`, where `t` is a value of type `T`;
  3. The overload candidate sets for call `T::f`, where `T` is a companion object of type `T`.

  Callable references to members of companion objects are deprioritized, as you could always use the `T.Companion::f` syntax to reference them.

  Important: when building the OCS for a callable reference, `invoke` operator convention does not apply, and all property references are treated equally as function references, being placed in the same sets. For example, consider the following code:

```
fun foo() = 1
val foo = 2
...
val y = ::foo
```

  Here both function `foo` and property `foo` are valid candidates for the callable reference and are placed *in the same candidate set*, thus producing an overload ambiguity. It is not important whether there is a suitable `invoke` operator available for the type of property `foo`.

  Example: consider the following two functions:

```
fun foo(i: Int): Int = 2          // (1)
fun foo(d: Double): Double = 2.0 // (2)
```

  In the following case:

```
val x: (Int) -> Int = ::foo
```

candidate (1) is picked, because (assuming CRT is the type of the callable reference) the constraint CRT <: FT(`kotlin.Int`) → `kotlin.Int` is built and only candidate (1) is applicable w.r.t. this constraint.

In another case:

```
fun bar(f: (Double) -> Double) {}

bar(::foo)
```

candidate (2) is picked, because (assuming CRT is the type of the callable reference) the constraint CRT <: FT(`kotlin.Double`) → `kotlin.Double` is built and only candidate (2) is applicable w.r.t. this constraint.

Please note that no bidirectional resolution is performed here as there is only one candidate for `bar`. If there were more than one candidate, the bidirectional resolution process would apply, possibly resulting in an overload resolution failure.

11.6.2 Bidirectional resolution for callable calls

If a callable reference (or several callable references) is itself an argument to an overloaded function call, the resolution process is performed for both callables *simultaneously*.

Assume we have a call `f(::g, b, c)`.

1. For each overload candidate `f`, a separate overload resolution process is completed as described in other parts of this section, up to the point of picking the most specific candidate. During this process, the only constraint for the callable reference `::g` is that it is an argument of a function type;
2. For the most specific candidate `f` found during the previous step, the overload resolution process for `::g` is performed as described here and the most specific candidate for `::g` is selected.

   Note: this may result in selecting the most specific candidate for `f` which has no available candidates for `::g`, meaning the bidirectional resolution process fails when resolving `::g`.

When performing bidirectional resolution for calls with multiple callable reference arguments, the algorithm is exactly the same, with each callable reference resolved separately in step 2. This ensures the overload resolution process for every callable being called is performed only once.

11.7 Type inference and overload resolution

Type inference in Kotlin is a very complicated process, and it is performed

after overload resolution is done; meaning type inference may not affect the way overload resolution candidate is picked in any way.

> Note: if we had allowed interdependence between type inference and overload resolution, we would have been able to create an infinitely oscillating behaviour, leading to an infinite compilation.
>
> Important: an exception to this limitation is when a lambda return type is used to refine function applicability. By limiting the scope of interdependence between type inference and overload resolution to a single step, we avoid creating an oscillating behaviour.

11.8 Conflicting overloads

In cases when it is known two callables are definitely interlinked in overload resolution (e.g., two member function-like callables declared in the same classifier), meaning they will always be considered together for overload resolution, Kotlin compiler performs *conflicting overload* detection for such callables.

Two callables `f` and `g` are *definitely interlinked* in overload resolution, if the following are true.

- `f` is not overriding `g` (and vice versa);
- `f` and `g` belong to the same level of c-level partition;
- `f` and `g` are declared in the same scope.

Different platform implementations may extend which callables are considered as definitely interlinked.

Two definitely interlinked callables `f` and `g` may create a *overload conflict*, if they could result in an overload ambiguity on most regular call sites.

To check whether such situation is possible, we compare `f` and `g` w.r.t. their applicability for a phantom call site with a fully specified argument list (i.e., with no used default arguments). If both `f` and `g` are equally or more specific to each other and neither of them is selected by the additional steps of MSC selection, we have an overload conflict.

Different platform implementations may extend which callables are considered as conflicting overloads.