Overload resolution

Kotlin supports *function overloading*, that is, the ability for several functions of the same name to coexist in the same scope, with the compiler picking the most suitable one when such a function is called. This section describes this mechanism in detail.

Intro

Unlike other object-oriented languages, Kotlin does not only have object methods, but also top-level functions, local functions, extension functions and function-like values, which complicate the overloading process quite a lot. Kotlin also has infix functions, operator and property overloading which all work in a similar, but subtly different way.

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 parameter 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 some syntactic scope according to the following rules:

- All receivers available in an outer scope are also available in the nested scope;
- In the scope of a classifier declaration, the following receivers are available:
  - The implicit `this` object of the declared type;
  - The companion object (if one exists) of this class;
  - The companion objects (if any exist) of all its superclasses;
- If a function or a property is an extension, `this` parameter of the extension is also available inside the extension declaration;
- The scope of a lambda expression, if it has an extension function type, contains `this` argument of the lambda expression.

TODO(If I'm a companion object, is a companion object of my supertype an implicit receiver for me or not?)

The available receivers are prioritized in the following way:

- The receivers provided in the most inner scope have higher priority;
- In a classifier body, the implicit `this` receiver has higher priority than any companion object receiver;

- Current class companion object receiver has higher priority than any of the base class companion objects.

The implicit receiver having the highest priority is also called the *default implicit receiver*. The default implicit receiver is available in the scope as `this`. Other available receivers may be accessed using [labeled this-expressions][This-expressions].

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

The forms of call-expression

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

- A fully-qualified call: `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()`.

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*), 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.

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 function named Y at its declaration site, but imported into the current scope using [a renaming import][Importing] as X;
  - A constructor of the type named X at its declaration site;
- Property-like callables, one of the following with an operator function called `invoke` available as member or extension in the current scope:
  - A property named X at its declaration site;
  - [An object][Object declarations] named X at its declaration site;
  - [A companion object][Companion objects] of a classifier 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][Importing] as X.

In the latter case a call $X(Y_0, Y_1, \ldots, Y_N)$ is an overloadable operator which is expanded to X.invoke$(Y_0, Y_1, \ldots, Y_N)$. 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 explicit receivers itself (denoted by a [`this`][This-expression] expression, labeled or not) 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`$(Y_0, Y_1, \ldots, Y_N)$ is an overloadable operator which is expanded to `this@A.invoke`$(Y_0, Y_1, \ldots, Y_N)$.

A *member callable* is either a member function-like callable or a member property-like callable with a member operator `invoke`. An *extension callable* is either an extension function-like callable, a member property-like callable with an extension operator `invoke` or an extension property-like callable with an extension operator `invoke`.

When calculating overload candidate sets, member callables produce the following separate sets (ordered by higher priority first):

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

Extension callables produce the following separate sets (ordered by higher priority first):

- Extension functions;
- 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 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.

Overload resolution for a fully-qualified call

If a callable name is fully-qualified (that is, it contains a full package path), then the overload candidate set S simply contains all the 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.

TODO(Clear up this mess)

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`.

A call with an explicit receiver

If a function call is done via a [navigation operator][Navigation operators] (`.` or `?.`, not to be confused with a fully-qualified call), then the left hand side operand of the call is the explicit receiver of this call.

A call of callable `f` with an explicit receiver `e` is correct if one (or more) of the following holds:

1. `f` is a member callable of the classifier type of `e` or any of its supertypes;
2. `f` is an extension callable of the classifier type of `e` or any of its supertypes, including local and imported extensions.

   Important: callables for case 2 include not only top-level extension callables, but also extension callables from any of the available implicit receivers. For example, if class P contains a member extension function for another class T and an object of class P is available as an implicit receiver, this extension function may be used for the call if it has a suitable type.

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

TODO(Sync with scopes and stuff when we have them)

1. The sets of non-extension member callables named `f` of type `T`;
2. The sets of local extension callables named `f`, whose receiver type conforms to type `T`, in all declaration scopes containing the current declaration scope, ordered by the size of the scope (smallest first), excluding the package scope;
3. The sets of explicitly imported extension callables named `f`, whose receiver type conforms to type `T`;
4. The sets of extension callables named `f`, whose receiver type conforms to type `T`, declared in the package scope;
5. The sets of star-imported extension callables named `f`, whose receiver type conforms to type `T`;

6. The sets of implicitly imported extension callables named `f`, whose receiver type conforms to type `T`.

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

When analyzing these sets, the **first** set that contains **any** callable with the corresponding name and conforming types is picked. This means, among other things, that if the set constructed on step 2 contains the overall most suitable candidate function, but the set constructed on step 1 is not empty, the functions from set 1 will be picked despite them being less suitable overload candidates.

Infix function calls

Infix function calls are a special case of function calls with an 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 functions considered for inclusion are the ones with the `infix` modifier. All other functions (and any properties) are not even considered for inclusion. Aside from this difference, candidates are selected using the same rules as for normal calls with explicit receiver.

> Note: this also means that all properties available through the `invoke` convention are non-eligible for infix calls, as there is no way of specifying the `infix` modifier for them.

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

Operator calls

According to [the overloadable operators section][Overloadable operators], some operator expressions in Kotlin can be overloaded using specially-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. The selection of an exact function called in each particular case is based on the same rules as for function calls with explicit receivers, the only difference being that only functions with `operator` modifier are considered for inclusion when building overload candidate sets. Any properties are never considered for the overload candidate sets of operator calls.

> 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` conventions 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 any calls arising from expanding [`for`-loop][For-loop statement] iteration conventions, [assignments][Assignments] or [property delegates][Delegated property declaration].

A call without an explicit receiver

A call which is performed with unqualified function name and without using a navigation operator is a call without an explicit receiver. It may have one or more implicit receivers or reference a top-level function.

> Note: this does not include calls using the `invoke` operator function where the left side of the call operator is not an identifier, but some other kind of expression. These cases are handled the same way as covered in the previous section and need no special treatment

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

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

1. The sets of local non-extension functions named `f` available in the current scope, in order of the scope they are declared in, smallest scope first;
2. The overload candidate sets for each implicit receiver `r` and `f`, calculated as if `r` is the explicit receiver, in order of the receiver priority (see the corresponding section);
3. Top-level non-extension functions named `f`, in the order of:
    a. Functions explicitely imported into current file;
    b. Functions declared in the same package;
    c. Functions star-imported into current file;
    d. Implicitly imported functions (either Kotlin standard library or platform-specific ones).

When analyzing these sets, the **first** set which contains **any** function with the corresponding name and conforming types is picked.

Calls with named parameters

Most of the 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 parameters for all named parameters from the 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 (see the MSC selection process) 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.

Calls with trailing lambda expressions

A call expression may have a single lambda expression placed outside of the argument list parentheses or even completely replacing them (see [this section][Call expression] for further details). This has no effect on the overload resolution process, aside from the argument reordering which may happen because of variable argument 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`.

Calls with specified type parameters

A call expression may have a type argument list explicitly specified before the argument list (see [this section][Call expression] for further details).. In this case all the potential overload sets only include callables which contain exactly the same number of formal type parameters at declaration site. In case of a property callable via `invoke` convention, type parameters must be present at the `invoke` operator function declaration.

Determining function applicability for a specific call

Rationale

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

Description

Determining function applicability for a specific call is a [type constraint][Type constraints] problem. First, for every non-lambda argument of the function

supplied in the call, 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 parameter inferred to have type T_i, corresponding to the function argument 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 parameter with the number of lambda arguments known to be K, corresponding to the function argument of type U_m, a special constraint of the form $R(L_1, \ldots, L_K) <: U_m$ is added to the constraint system, where $R, L_1, \ldots, L_K$ are fresh variables;
- For each lambda parameter with an unknown number of lambda arguments (that is, being equal to 0 or 1), a special constraint of the form $kotlin.Function <: U_m$ is added to the constraint system, where $kotlin.Function$ is the common base of all functional types .

(TODO(what's the spec name?))

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

Choosing the most specific function from the overload candidate set

Rationale

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

> The most specific function can forward itself to any other function 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 ambiguity error should be reported by the compiler.

Consider the following example with two functions:

```
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.

The following snippet should explain this in more detail.

```
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 try to clarify this mechanism in more detail.

Description

When an overload resolution set S is selected and it contains more than one callable, the next step is to choose the most appropriate candidate from these callables.

The selection process uses the [type constraint][Type constraints] system 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, the corresponding value parameter types $X_1, X_2, X_3, \ldots, X_N$ of F_1 and $Y_1, Y_2, Y_3, \ldots, Y_N$ of F_2, a type constraint $X_K <: Y_K$ is built **unless both X_K and Y_K are [built-in integer types][Built-in integer types].** During construction of these constraints, all type parameters $T_1, T_2, \ldots, T_M$ of F_1 are considered bound to fresh type variables $T_1^{\sim}, T_2^{\sim}, \ldots, T_M^{\sim}$, and all type parameters of F_2 are considered free;
- All declaration-site type constraints of $X_1, X_2, X_3, \ldots, X_N$ and $Y_1, Y_2, Y_3, \ldots, Y_N$ are also added to the constraint system.

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. If F_1 is equally or more applicable than F_2 and F_2 is equally or more applicable than F_1, this means that the two callables are equally applicable and additional decision steps are needed to choose the most specific overload candidate. If neither F_1 nor F_2 is equally or more applicable than its counterpart, it also means that F_1 and F_2 are equally applicable and additional decision steps are needed.

All members of the overload candidate set are ranked according to the applicability criteria. If there are several callables which are more applicable than all other candidates and equally applicable to each other, an additional step is performed.

- Any non-generic (meaning that it does not have type parameters in its declaration) callable is a more specific candidate than any generic (containing type parameters in its declaration) callable. If there are several non-generic candidates, further steps are limited to those candidates;

- For every non-default argument of the call consider the corresponding value parameter types $X_1, X_2, X_3, \ldots, X_N$ of F_1 and $Y_1, Y_2, Y_3, \ldots, Y_N$ of F_2. If, for any K, both X_K and Y_K are different built-in integer types and one of them is `kotlin.Int`, then this parameter is preferred over the other parameter of the call. If all such parameters of F_1 are preferred based on this criteria over the parameters of F_2, then F_1 is a more specific candidate than F_2, and vice versa.
- 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;
- If the number of non-specified default parameters is equal for several 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 important in cases where the actual call argument is an integer literal having an [integer literal type][Integer literal types]. In this particular case, several functions with different built-in integer types for the corresponding parameter may be applicable, and it is preferred to have the `kotlin.Int` overload as the most specific.

If after this additional step there are still several candidates that are equally applicable for the call, this is an **overload ambiguity** which must be reported as a compiler error.

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

About type inference

[Type inference][Type inference] in Kotlin is a pretty complicated process, which is performed after resolving all the overload candidates. Due to the complexity of the process, type inference may not affect the way overload resolution candidate is picked up.

TODOs

- Property business
- Function types (type system section?)
- Definition of “type parameter level”

- Calls with trailing lambda without parameter type
  - Lambdas with parameter types seem to be covered (**nope, they are not**)
- Calls with specified type parameters `f<Double>(3)`
- Widen the notion of "function" and "property" during overloading
  - Constructors and companion object `invoke` (clash with functions)
  - Singleton objects (clash with properties)
  - Enum constants (clash with properties)
  - Explicit `this` cannot clash with properties, but can clash with other explicit `this`, meaning it effectively overloads over all the available receivers in the scope
  - Can `super` be overloaded? I suppose