Kotlin type constraints

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

Type constraint definition

A *type constraint* in general is an inequation of the following form: $T <: U$ where T and U are Kotlin types (see [type system][Type system]). It is important, however, that Kotlin has parameterized types and type parameters of T and U (or type parameters of their parameters, or T and U themselves) may be *type variables*, that are unknown types that may be substituted by any other type in Kotlin.

Please note that, in general, type variables of the constraint system are not the same as type parameters of a type or a callable. Some type parameters may be *bound* in the constraint system, meaning that, although they are not known yet in Kotlin code, they are not type variables and are not to be substituted.

When such an ambiguity arises, we will use the notation T_i for a type variable and $\tilde{T}_i$ for a bound type parameter. The main difference between bound parameters and concrete types is that different concrete types may not be equal, but a bound parameter may be equal to another bound parameter or a concrete type.

Several examples of valid type constraints:

- $\texttt{List}\left\langle \tilde{X} \right\rangle <: Y$
- $\texttt{List}\left\langle \tilde{X} \right\rangle <: \texttt{List}\left\langle \texttt{List}\left\langle \texttt{Int} \right\rangle \right\rangle$
- $\widetilde{X} <: Y$

Every constraint system has implicit constraints $\texttt{Any} <: T_j$ and $T_j <: \texttt{Nothing}?$ for every type T_j mentioned in constraint, including type variables.

Type constraint solving

There are two tasks that a type constraint solver may perform: checking constraint system for soundness and solving the system, e.g. inferring viable values for all the type variables that have themselves no type variables in them.

Checking a constraint system for soundness can be viewed as a simpler case of solving a constraint, as if there is a solution, then the system is sound. It is, however, a much simpler task with only two possible outcomes. Solving a constraint system, on the other hand, may have several different results as there may be several valid solutions.

Constraint examples that are sound yet no relevant solutions exist:

- $X <: Y$
- $\texttt{List}\,\langle X\rangle <: \texttt{Collection}\,\langle X\rangle$

Checking constraint system soundness

TODO()

Finding optimal solution

As any constraint system may have several valid solutions, finding one that is "optimal" in some sense is not possible in general, because the notion of the best solution for a task depends on a particular use-case. To solve this problem, the constraint system allows two additional types of constraints:

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

If a variable have no constraints of these two kinds associated with it, it is assumed to have a pull-up constraint, that is, in an ambigious situation, the biggest possible type is chosen.

TODO()

The relations on types as constraints

In the other chapters (see [expressions] and [statements] for example) the relations between types may be expressed using the type operations found in the [type system section][Type system] of this document. Not all of these relations are easily converted into their constraint form.

The greatest lower bound of two types is converted directly, as the greatest lower bound is always an intersection type. The least upper bound, however, is a little bit tricky. If type T is defined to be the least upper bound of A and B with all these types being either known, unknown or containing type variables, the following constraints are produced:

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

- $\uparrow B$

  Example:
  Let's assume we have the following code:

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

  where `a`, `b`, `c` are some expressions with types completely unknown (having no other type constraints besided the implicit ones). Let's assume the type variables generated for them to be A, B and C respectively and the type variable for `e` being E. This, according to [the conditional expression chapter][Conditional expression], produces the following relations:

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

  These, in turn, produce the following constraints (here we omit the implicit relations of all type variables with `kotlin.Any?` and `kotlin.Nothing`):

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

  Which, according to the semantics of additional constraints (and the default pull-up constraint for C), produce the following solution:

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

TODO(prove that these constraints are equivalent to LUB from type system?)

TODO(are they actually?)

TODO(does that matter?)