Chapter 3

Types

```
Type                  ::=  FunType
                        |  TypeLambda
                        |  InfixType
FunType               ::=  FunTypeArgs ‘=>’ Type
                        |  TypeLambdaParams '=>' Type
TypeLambda            ::=  TypeLambdaParams ‘=>>’ Type
InfixType             ::=  RefinedType
                        |  RefinedTypeOrWildcard id [nl]
RefinedTypeOrWildcard {id [nl] RefinedTypeOrWildcard}
RefinedType           ::=  AnnotType {[nl] Refinement}
AnnotType             ::=  SimpleType {Annotation}
SimpleType            ::=  SimpleLiteral
                        |  SimpleType1
SimpleType1           ::=  id
                        |  Singleton ‘.’ id
                        |  Singleton ‘.’ ‘type’
                        |  ‘(’ TypesOrWildcards ‘)’
                        |  Refinement
                        |  SimpleType1 TypeArgs
                        |  SimpleType1 ‘#’ id
Singleton             ::=  SimpleRef
                        |  SimpleLiteral
                        |  Singleton ‘.’ id
SimpleRef             ::=  id
                        |  [id ‘.’] ‘this’
                        |  [id ‘.’] ‘super’ [‘[’ id ‘]’] ‘.’ id
ParamType             ::=  [‘=>’] ParamValueType
ParamValueType        ::=  ParamValueType [‘*’]
TypeArgs              ::=  ‘[’ TypesOrWildcards ‘]’
Refinement            ::=  :<<< [RefineDcl] {semi [RefineDcl]} >>>

FunTypeArgs           ::=  InfixType
                        |  ‘(’ [ FunArgTypes ] ‘)’
                        |  FunParamClause
FunArgTypes           ::=  FunArgType { ‘,’ FunArgType }
FunArgType            ::=  Type
                        |  ‘=>’ Type
FunParamClause        ::=  ‘(’ TypedFunParam {‘,’ TypedFunParam } ‘)’
TypedFunParam         ::=  id ‘:’ Type

TypeLambdaParams      ::=  ‘[’ TypeLambdaParam {‘,’ TypeLambdaParam} ‘]’
TypeLambdaParam       ::=  {Annotation} (id | ‘_’) [TypeParamClause]
TypeLambdaParam       ::=  {Annotation} (id | ‘_’) [TypeParamClause]
TypeBounds
TypeParamClause       ::=  ‘[’ VariantTypeParam {‘,’ VariantTypeParam}
‘]’
VariantTypeParam      ::=  {Annotation} [‘+’ | ‘-’] (id | ‘_’)
[TypeParamClause] TypeBounds

RefineDcl             ::=  ‘val’ ValDcl
                        |  ‘def’ DefDcl
                        |  ‘type’ {nl} TypeDcl

TypeBounds            ::=  [‘>:’ Type] [‘<:’ Type]

TypesOrWildcards      ::=  TypeOrWildcard {‘,’ TypeOrWildcard}
TypeOrWildcard        ::=  Type
                        |  WildcardType
RefinedTypeOrWildcard ::=  RefinedType
                        |  WildcardType
WildcardType          ::=  (‘?‘ | ‘_‘) TypeBounds
```

The above grammer describes the concrete syntax of types that can be written in user code. Semantic operations on types in the Scala type system are better defined in terms of *internal types*, which are desugared from the concrete type syntax.

3.1 Internal Types

The following *abstract grammar* defines the shape of *internal types*. In this specification, unless otherwise noted, "types" refer to internal types. Internal types abstract away irrelevant details such as precedence and grouping, and contain shapes of types that cannot be directly expressed using the concrete syntax. They also contain simplified, decomposed shapes for complex concrete syntax types, such as refined types.

```
Type              ::=  ‘AnyKind‘
                    |  ‘Nothing‘
                    |  TypeLambda
                    |  DesignatorType
                    |  ParameterizedType
                    |  ThisType
                    |  SuperType
                    |  LiteralType
                    |  ByNameType
                    |  AnnotatedType
                    |  RefinedType
                    |  RecursiveType
                    |  RecursiveThis
                    |  UnionType
                    |  IntersectionType
                    |  SkolemType


TypeLambda        ::=  ‘[‘ TypeParams ‘]‘ ‘=>>‘ Type
TypeParams        ::=  TypeParam {‘,‘ TypeParam}
TypeParam         ::=  ParamVariance id TypeBounds
ParamVariance     ::=  ε | ‘+‘ | ‘-‘

DesignatorType    ::=  Prefix ‘.‘ id
Prefix            ::=  Type
                    |  PackageRef
                    |  ε
PackageRef        ::=  id {‘.‘ id}

ParameterizedType ::=  Type ‘[‘ TypeArgs ‘]‘
TypeArgs          ::=  TypeArg {‘,‘ TypeArg}
TypeArg           ::=  Type
                    |  WilcardTypeArg
WildcardTypeArg   ::=  ‘?‘ TypeBounds

ThisType          ::=  classid ‘.‘ ‘this‘
SuperType         ::=  classid ‘.‘ ‘super‘ ‘[‘ classid ‘]‘
LiteralType       ::=  SimpleLiteral
ByNameType        ::=  ‘=>‘ Type
AnnotatedType     ::=  Type Annotation

RefinedType       ::=  Type ‘{‘ Refinement ‘}‘
Refinement        ::=  ‘type‘ id TypeAliasOrBounds
                    |  ‘def‘ id ‘:‘ TypeOrMethodic
                    |  ‘val‘ id ‘:‘ Type

RecursiveType     ::=  ‘{‘ recid ‘=>‘ Type ‘}‘
RecursiveThis     ::=  recid ‘.‘ ‘this‘

UnionType         ::=  Type ‘|‘ Type
IntersectionType  ::=  Type ‘&‘ Type

SkolemType        ::=  ‘∃‘ skolemid ‘:‘ Type

TypeOrMethodic    ::=  Type
                    |  MethodicType
MethodicType      ::=  MethodType
                    |  PolyType

MethodType        ::=  ‘(‘ MethodTypeParams ‘)‘ TypeOrMethodic
MethodTypeParams  ::=  ε
                    |  MethodTypeParam {‘,‘ MethodTypeParam}
MethodTypeParam   ::=  id ‘:‘ Type

PolyType          ::=  ‘[‘ PolyTypeParams ‘]‘ TypeOrMethodic
PolyType          ::=  ‘[‘ PolyTypeParams ‘]‘ TypeOrMethodic

PolyTypeParams    ::=  PolyTypeParam {‘,‘ PolyTypeParam}
PolyTypeParam     ::=  ‘id‘ TypeBounds

TypeAliasOrBounds ::=  TypeAlias
                    |  TypeBounds
TypeAlias         ::=  ‘=‘ Type
TypeBounds        ::=  ‘<:‘ Type ‘>:‘ Type
```

3.1.1 Translation of Concrete Types into Internal Types

Concrete types are recursively translated, or desugared, into internal types. Most shapes of concrete types have a one-to-one translation to shapes of internal types. We elaborate hereafter on the translation of the other ones.

3.1.2 Infix Types

```
InfixType     ::=  CompoundType {id [nl] CompoundType}
```

A concrete *infix type* T_1 op T_2 consists of an infix operator op which gets applied to two type operands T_1 and T_2. The type is translated to the internal type application op$[T_1, T_2]$. The infix operator op may be an arbitrary identifier.

Type operators follow the same precedence and associativity as term operators. For example, `A + B * C` parses as `A + (B * C)` and `A | B & C` parses as `A | (B & C)`. Type operators ending in a colon ‘:’ are right-associative; all other operators are left-associative.

In a sequence of consecutive type infix operations $t_0 \; op \; t_1 \; op_2 \ldots op_n \; t_n$, all operators $op_1, \ldots, op_n$ must have the same associativity. If they are all left-associative, the sequence is interpreted as $(\ldots(t_0 \, op_1 t_1) \, op_2 \ldots) \, op_n t_n$, otherwise it is interpreted as $t_0 \, op_1 (t_1 \, op_2 (\ldots op_n t_n) \ldots)$.

The type operators `|` and `&` are not really special. Nevertheless, unless shadowed, they resolve to `scala.|` and `scala.&`, which represent union and intersection types, respectively.

3.1.3 Function Types

```
Type              ::=  FunTypeArgs ‘=>’ Type
FunTypeArgs       ::=  InfixType
                    |  ‘(’ [ FunArgTypes ] ‘)’
                    |  FunParamClause
FunArgTypes       ::=  FunArgType { ‘,’ FunArgType }
FunArgType        ::=  Type
                    |  ‘=>’ Type
FunParamClause    ::=  ‘(’ TypedFunParam {‘,’ TypedFunParam } ‘)’
TypedFunParam     ::=  id ‘:’ Type
```

```
  TypedFunParam     ::=  id  ':'  Type
```

The concrete function type $(T_1, \ldots, T_n) \Rightarrow R$ represents the set of function values that take arguments of types $T_1, \ldots, Tn$ and yield results of type R. The case of exactly one argument type $T \Rightarrow R$ is a shorthand for $(T) \Rightarrow R$. An argument type of the form $\Rightarrow T$ represents a call-by-name parameter of type T.

Function types associate to the right, e.g. $S \Rightarrow T \Rightarrow R$ is the same as $S \Rightarrow (T \Rightarrow R)$.

Function types are covariant in their result type and contravariant in their argument types.

Function types translate into internal class types that define an `apply` method. Specifically, the n-ary function type $(T_1, \ldots, T_n) \Rightarrow R$ translates to the internal class type `scala.Function`$_n$`[`T_1`, ..., `T_n`, `R`]`. In particular $() \Rightarrow R$ is a shorthand for class type `scala.Function`$_0$`[`R`]`.

Such class types behave as if they were instances of the following trait:

```
trait Function$_n$[-$T_1$, ..., -$T_n$, +$R$]:
  def apply($x_1$: $T_1$, ..., $x_n$: $T_n$): $R$
```

Their exact supertype and implementation can be consulted in the function classes section of the standard library page in this document.

Dependent function types are function types whose parameters are named and can referred to in result types. In the concrete type $(x_1 : T_1, \ldots, x_n : T_n) \Rightarrow R$, R can refer to the parameters x_i, notably to form path-dependent types. It translates to the internal refined type

```
scala.Function$_n$[$T_1$, ..., $T_n$, $S$] {
  def apply($x_1$: $T_1$, ..., $x_n$: $T_n$): $R$
}
```

where S is the least super type of R that does not mention any of the x_i.

Polymorphic function types are function types that take type arguments. Their result type must be a function type. In the concrete type $[a_1 >: L_1 <: H_1, \ldots, a_n >: L_1 <: H_1] => (T_1, \ldots, T_m) => R$, the types T_j and R can refer to the type parameters a_i. It translates to the internal refined type

```
scala.PolyFunction {
  def apply[$a_1 >: L_1 <: H_1, \ldots, a_n >: L_1 <: H_1$]($x_1$: $T_1$, ..., $x_n$: $T_n$): $R$
}
```

3.1.4 Concrete Refined Types

```
RefinedType           ::=  AnnotType {[nl] Refinement}
SimpleType1           ::=  ...
                        |  Refinement
Refinement            ::=  :<<< [RefineDcl] {semi [RefineDcl]} >>>

RefineDcl             ::=  ‘val’ ValDcl
                        |  ‘def’ DefDcl
                        |  ‘type’ {nl} TypeDcl
```

In the concrete syntax of types, refinements can contain several refined declarations. Moreover, the refined declarations can refer to each other as well as to members of the parent type, i.e., they have access to `this`.

In the internal types, each refinement defines exactly one refined declaration, and references to `this` must be made explicit in a recursive type.

The conversion from the concrete syntax to the abstract syntax works as follows:

1. Create a fresh recursive this name α.
2. Replace every implicit or explicit reference to `this` in the refinement declarations by α.
3. Create nested refined types, one for every refined declaration.
4. Unless α was never actually used, wrap the result in a recursive type `{` α `=> ... }`.

3.1.5 Concrete Type Lambdas

```
TypeLambda            ::= TypeLambdaParams ‘=>>’ Type
TypeLambdaParams      ::=  ‘[’ TypeLambdaParam {‘,’ TypeLambdaParam} ‘]’
TypeLambdaParam       ::=  {Annotation} (id | ‘_’) [TypeParamClause]
TypeBounds
TypeParamClause       ::=  ‘[’ VariantTypeParam {‘,’ VariantTypeParam}
‘]’
VariantTypeParam      ::=  {Annotation} [‘+’ | ‘-’] (id | ‘_’)
[TypeParamClause] TypeBounds
```

At the top level of concrete type lambda parameters, variance annotations are not allowed. However, in internal types, all type lambda parameters have explicit variance annotations.

When translating a concrete type lambda into an internal one, the variance of each type parameter is *inferred* from its usages in the body of the type lambda.

3.2 Definitions

From here onwards, we refer to internal types by default.

3.2.1 Kinds

The Scala type system is fundamentally higher-kinded. *Types* are either *proper types*, *type constructors* or *poly-kinded types*.

- Proper types are the types of *terms*.
- Type constructors are type-level functions from types to types.
- Poly-kinded types can take various kinds.

All types live in a single lattice with respect to a *conformance* relationship $<:$. The *top type* is `AnyKind` and the *bottom type* is `Nothing`: all types conform to `AnyKind`, and `Nothing` conforms to all types. They can be referred to as the standard library entities `scala.AnyKind` and `scala.Nothing`, respectively.

Types can be *concrete* or *abstract*. An abstract type T always has lower and upper bounds L and H such that $L >: T$ and $T <: H$. A concrete type T is considered to have itself as both lower and upper bound.

The kind of a type is indicated by its (transitive) upper bound:

- A type $T <:$ `scala.Any` is a proper type.
- A type $T <: K$ where K is a *type lambda* (of the form `[`$\pm a_1 >: L_1 <: H_1$`, ..., `$\pm a_n >: L_n <: H_n$`] =>> `U) is a type constructor.
- Other types are poly-kinded; they are neither proper types nor type constructors.

As a consequece, `AnyKind` itself is poly-kinded. `Nothing` is *universally-kinded*: it has all kinds at the same time, since it conforms to all types.

With this representation, it is rarely necessary to explicitly talk about the kinds of types. Usually, the kinds of types are implicit through their bounds.

Another way to look at it is that type bounds *are* kinds. They represent sets of types: $>: L <: H$ denotes the set of types T such that $L <: T$ and $T <: H$. A set of types can be seen as a *type of types*, i.e., as a *kind*.

Conventions

Type bounds are formally always of the form $>: L <: H$. By convention, we can omit either of both bounds in writing.

- When omitted, the lower bound L is `Nothing`.
- When omitted, the higher bound H is `Any` (*not* `AnyKind`).

These conventions correspond to the defaults in the concrete syntax.

3.2.2 Proper Types

Proper types are also called *value types*, as they represent sets of *values*.

Stable types are value types that contain exactly one non-`null` value. Stable types can be used as prefixes in named designator types. The stable types are

- designator types referencing a stable term,
- this types,
- super types,
- literal types,
- recursive this types, and
- skolem types.

Every stable type T is concrete and has an *underlying* type U such that $T <: U$.

3.2.3 Type Constructors

To each type constructor corresponds an *inferred type parameter clause* which is computed as follows:

- For a type lambda, its type parameter clause (including variance annotations).
- For a polymorphic class type, the type parameter clause of the referenced class definition.
- For a non-class type designator, the inferred clause of its upper bound.

3.2.4 Type Definitions

A *type definition* D represents the right-hand-side of a `type` declaration or the bounds of a type parameter. It is either:

- a type alias of the form $= U$, or
- an abstract type definition with bounds $>: L <: H$.

All type definitions have a lower bound L and an upper bound H, which are types. For type aliases

All type definitions have a lower bound L and an upper bound H, which are types. For type aliases,

$L = H = U$.

The type definition of a type parameter is never a type alias.

3.3 Types

3.3.1 Type Lambdas

```
TypeLambda     ::=  ‘[‘ TypeParams ‘]‘ ‘=>>‘ Type
TypeParams     ::=  TypeParam {‘,‘ TypeParam}
TypeParam      ::=  ParamVariance id TypeBounds
ParamVariance  ::=  ε | ‘+‘ | ‘-‘
```

A *type lambda* of the form $[\pm a_1 >: L_1 <: H_1, \; ..., \; \pm a_n >: L_n <: H_n]$ =>> U is a direct representation of a type constructor with n type parameters. When applied to n type arguments that conform to the specified bounds, it produces another type U. Type lambdas are always concrete types.

All type constructors conform to some type lambda.

The type bounds of the parameters of a type lambda are in contravariant position, while its result type is in covariant position. If some type constructor $T <: [\pm a_1 >: L_1 <: H_1, \; ..., \; \pm a_n >: L_n <: H_n]$ =>> U, then T's ith type parameter bounds contain the bounds $>: L_i <: H_i$, and its result type conforms to U.

Note: the concrete syntax of type lambdas does not allow to specify variances for type parameters. Instead, variances are inferred from the body of the lambda to be as general as possible.

Example

```
type Lst = [T] =>> List[T] // T is inferred to be covariant with bounds
>: Nothing <: Any
type Fn = [A <: Seq[?], B] =>> (A => B) // A is inferred to be
contravariant, B covariant

val x: Lst[Int] = List(1) // ok, Lst[Int] expands to List[Int]
val f: Fn[List[Int], Int] = (x: List[Int]) => x.head // ok

val g: Fn[Int, Int] = (x: Int) => x // error: Int does not conform to the
bound Seq[?]

def liftPair[F <: [T] =>> Any](f: F[Int]): Any = f
liftPair[Lst](List(1)) // ok, Lst <: ([T] =>> Any)
```

3.3.2 Designator Types

```
DesignatorType    ::=  Prefix ‘.‘ id
Prefix            ::=  Type
                    |  PackageRef
                    |  ε
PackageRef        ::=  id {‘.‘ id}
```

A designator type (or designator for short) is a reference to a definition. Term designators refer to term definitions, while type designators refer to type definitions.

In the abstract syntax, the `id` retains whether it is a term or type. In the concrete syntax, an `id` refers to a *type* designator, while `id.type` refers to a *term* designator. In that context, term designators are often called *singleton types*.

Designators with an empty prefix ϵ are called direct designators. They refer to local definitions available in the scope:

- Local `type`, `object`, `val`, `lazy val`, `var` or `def` definitions
- Term or type parameters

The `id`s of direct designators are protected from accidental shadowing in the abstract syntax. They retain the identity of the exact definition they refer to, rather than relying on scope-based name resolution. [1]

The ϵ prefix cannot be written in the concrete syntax. A bare `id` is used instead and resolved based on scopes.

Named designators refer to *member* definitions of a non-empty prefix:

- Top-level definitions, including top-level classes, have a package ref prefix
- Class member definitions and refinements have a type prefix

Term Designators

A term designator $p.x$ referring to a term definition `t` has an *underlying type* U. If $p = \epsilon$ or p is a package ref, the underlying type U is the *declared type* of `t` and $p.x$ is a stable type if an only if `t` is a `val` or `object` definition. Otherwise, the underlying type U and whether $p.x$ is a stable type are determined by `memberType`(p, x).

All term designators are concrete types. If `scala.Null` $<:$ U, the term designator denotes the set of values consisting of `null` and the value denoted by t, i.e., the value v for which `t eq v`. Otherwise, the designator denotes the singleton set only containing v.

Type Designators

A type designator $p.C$ referring to a *class* definition (including traits and hidden object classes) is a *class type*. If the class is monomorphic, the type designator is a value type denoting the set of instances of C or any of its subclasses. Otherwise it is a type constructor with the same type parameters as the class definition. All class types are concrete, non-stable types.

If a type designator $p.T$ is not a class type, it refers to a type definition `T` (a type parameter or a `type` declaration) and has an *underlying* *type definition*. If $p = \epsilon$ or p is a package ref, the underlying type definition is the *declared type definition* of `T`. Otherwise, it is determined by `memberType(`p`, `T`)`. A non-class type designator is concrete (resp. stable) if and only if its underlying type definition is an alias U and U is itself concrete (resp. stable).

3.3.3 Parameterized Types

```
ParameterizedType ::=  Type ‘[‘ TypeArgs ‘]‘
TypeArgs          ::=  TypeArg {‘,‘ TypeArg}
TypeArg           ::=  Type
                    |  WilcardTypeArg
WildcardTypeArg   ::=  ‘?‘ TypeBounds
```

A *parameterized type* $T[T_1, ..., T_n]$ consists of a type constructor T and type arguments $T_1, ..., T_n$ where $n \geq 1$. The parameterized type is well-formed if

- T is a type constructor which takes n type parameters $a_1, ..., a_n$, i.e., it must conform to a type lambda of the form $[\pm a_1 >: L_1 <: H_1, ..., \pm a_n >: L_n <: H_n] => U$, and
- if T is an abstract type constructor, none of the type arguments is a wildcard type argument, and
- each type argument *conforms to its bounds*, i.e., given σ the substitution $[a_1 := T_1, ..., a_n := T_n]$, for each type i:
    - if T_i is a type and $\sigma L_i <: T_i <: \sigma H_i$, or
    - T_i is a wildcard type argument $? >: L_{Ti} <: H_{Ti}$ and $\sigma L_i <: L_{Ti}$ and $H_{Ti} <: \sigma H_i$.

$T[T_1, ..., T_n]$ is a *parameterized class type* if and only if T is a class type. All parameterized class types are value types.

In the concrete syntax of wildcard type arguments, if both bounds are omitted, the real bounds are inferred from the bounds of the corresponding type parameter in the target type constructor (which must be concrete). If only one bound is omitted, `Nothing` or `Any` is used, as usual.

Simplification Rules

Wildcard type arguments used in covariant or contravariant positions can always be simplified to regular types.

Let $T[T_1, ..., T_n]$ be a parameterized type for a concrete type constructor. Then, applying a wildcard type argument $? >: L <: H$ at the i'th position obeys the following equivalences:

- If the type parameter T_i is declared covariant, then $T[..., ? >: L <: H, ...] =:= T[..., H, ...]$.
- If the type parameter T_i is declared contravariant, then $T[..., ? >: L <: H, ...] =:= T[..., L, ...]$.

Example Parameterized Types

Given the partial type definitions:

```
class TreeMap[A <: Comparable[A], B] { ... }
class List[+A] { ... }
class I extends Comparable[I] { ... }

class F[M[A], X] { ... } // M[A] desugars to M <: [A] =>> Any
class S[K <: String] { ... }
class G[M[Z <: I], I] { ... } // M[Z <: I] desugars to M <: [Z <: I] =>>
Any
```

the following parameterized types are well-formed:

```
TreeMap[I, String]
List[I]
List[List[Boolean]]

F[List, Int]
F[[X] =>> List[X], Int]
G[S, String]

List[?] // ? inferred as List[_ >: Nothing <: Any], equivalent to
List[Any]
List[? <: String] // equivalent to List[String]
S[? <: String]
F[?, Boolean] // ? inferred as ? >: Nothing <: [A] =>> Any
```

and the following types are ill-formed:

```
TreeMap[I]            // illegal: wrong number of parameters
TreeMap[List[I], Int] // illegal: type parameter not within bound
List[[X] => List[X]]

F[Int, Boolean]       // illegal: Int is not a type constructor
```

```
F[TreeMap, Int]       // illegal: TreeMap takes two parameters,
                      //   F expects a constructor taking one
F[[X, Y] => (X, Y)]
G[S, Int]             // illegal: S constrains its parameter to
                      //   conform to String,
                      // G expects type constructor with a parameter
                      //   that conforms to Int
```

The following code also contains an ill-formed type:

```
trait H[F[A]]:  // F[A] desugars to F <: [A] =>> Any, which is abstract
  def f: F[_]   // illegal : an abstract type constructor
                // cannot be applied to wildcard arguments.
```

3.3.4 This Types

```
ThisType  ::=  classid ‘.‘ ‘this‘
```

A *this type* C`.this` denotes the `this` value of class C within C.

This types often appear implicitly as the prefix of designator types referring to members of C. They play a particular role in the type system, since they are affected by the as seen from operation on types.

This types are stable types. The underlying type of C`.this` is the self type of C.

3.3.5 Super Types

```
SuperType  ::=  classid ‘.‘ ‘super‘ ‘[‘ classid ‘]‘
```

A *super type* C`.super[`D`]` denotes the `this` value of class `C` within `C`, but "widened" to only see members coming from a parent class or trait D.

Super types exist for compatibility with Scala 2, which allows shadowing of inner classes. In a Scala 3-only context, a super type can always be replaced by the corresponding this type. Therefore, we omit further discussion of super types in this specification.

3.3.6 Literal Types

```
LiteralType  ::=  SimpleLiteral
```

A literal type `lit` denotes the single literal value `lit`. Thus, the type ascription `1: 1` gives the most precise type to the literal value `1`: the literal type `1`.

At run time, an expression `e` is considered to have literal type `lit` if `e == lit`. Concretely, the result of `e.isInstanceOf[lit]` and `e match { case _ : lit => }` is determined by evaluating `e == lit`.

Literal types are available for all primitive types, as well as for `String`. However, only literal types for `Int`, `Long`, `Float`, `Double`, `Boolean`, `Char` and `String` can be expressed in the concrete syntax.

Literal types are stable types. Their underlying type is the primitive type containing their value.

Example

```
val x: 1 = 1
val y: false = false
val z: false = y
val int: Int = x

val badX: 1 = int       // error: Int is not a subtype of 1
val badY: false = true  // error: true is not a subtype of false
```

3.3.7 By-Name Types

```
ByNameType  ::=  ‘=>‘ Type
```

A by-name type $=> T$ denotes the declared type of a by-name term parameter. By-name types can only appear as the types of parameters in method types, and as type arguments in parameterized types.

3.3.8 Annotated Types

```
AnnotatedType  ::=  Type Annotation
```

An *annotated type* $T a$ attaches the annotation a to the type T.

Example

The following type adds the `@suspendable` annotation to the type `String`:

```
String @suspendable
```

3.3.9 Refined Types

```
RefinedType  ::=  Type ‘{‘ Refinement ‘}‘
Refinement   ::=  ‘type‘ id TypeAliasOrBounds
```

```
                 |  ‘def‘ id ‘:‘ TypeOrMethodic
                 |  ‘val‘ id ‘:‘ Type
```

A *refined type* $T R$ denotes the set of values that belong to T and also have a *member* conforming to the refinement R.

The refined type $T R$ is well-formed if:

- T is a proper type, and
- if R is a term (`def` or `val`) refinement, the refined type is a proper type, and
- if R overrides a member of T, the usual rules for overriding apply, and
- if R is a `def` refinement with a polymorphic method type, then R overrides a member definition of T.

As an exception to the last rule, a polymorphic method type refinement is allowed if $T <:$ `scala.PolyFunction` and id is the name `apply`.

If the refinement R overrides no member of T and is not an occurrence of the `scala.PolyFunction` exception, the refinement is said to be “structural” [2].

Note: since a refinement does not define a *class*, it is not possible to use a this type to reference term and type members of the parent type T within the refinement. When the surface syntax of refined types makes such references, a recursive type wraps the refined type, given access to members of self through a recursive-this type.

Example

Given the following class definitions:

```scala
trait T:
  type X <: Option[Any]
  def foo: Any
  def fooPoly[A](x: A): Any

trait U extends T:
  override def foo: Int
  override def fooPoly[A](x: A): A

trait V extends T
  type X = Some[Int]
  def bar: Int
  def barPoly[A](x: A): A
```

We get the following conformance relationships:

- `U <: T { def foo: Int }`
- `U <: T { def fooPoly[A](x: A): A }`
- `U <: (T { def foo: Int }) { def fooPoly[A](x: A): A }` (we can chain refined types to refine multiple members)
- `V <: T { type X <: Some[Any] }`
- `V <: T { type X >: Some[Nothing] }`
- `V <: T { type X = Some[Int] }`
- `V <: T { def bar: Any }` (a structural refinement)

The following refined types are not well-formed:

- `T { def barPoly[A](x: A): A }` (structural refinement for a polymorphic method type)
- `T { type X <: List[Any] }` (does not satisfy overriding rules)
- `List { def head: Int }` (the parent type `List` is not a proper type)
- `T { def foo: List }` (the refined type `List` is not a proper type)
- `T { def foo: T.this.X }` (`T.this` is not allowed outside the body of `T`)

3.3.10 Recursive Types

```
RecursiveType  ::=  ‘{‘ recid ‘=>‘ Type ‘}‘
RecursiveThis  ::=  recid ‘.‘ ‘this‘
```

A *recursive type* of the form `{` α `=>` T `}` represents the same values as T, while offering T access to its *recursive this* type α.

Recursive types cannot directly be expressed in the concrete syntax. They are created as needed when a refined type in the concrete syntax contains a refinement that needs access to the `this` value. Each recursive type defines a unique self-reference α, distinct from any other recursive type in the system.

Recursive types can be unfolded during subtyping as needed, replacing references to its α by a stable reference to the other side of the conformance relationship.

Example

Given the class definitions in the refined types section, we can write the following refined type in the source syntax:

```
T { def foo: X }
// equivalent to
T { def foo: this.X }
```

This type is not directly expressible as a refined type alone, as the refinement cannot access the `this` value. Instead, in the abstract syntax of types, it is translated to `{ `α` => `T` { def foo: `α`.X } }`.

Given the following definitions:

```
trait Z extends T:
  type X = Option[Int]
  def foo: Option[Int] = Some(5)

val z: Z
```

we can check that `z `$<:$` { `α` => `T` { def foo: `α`.X } }`. We first unfold the recursive type, substituting z for α, resulting in `z `$<:$` T { def foo: z.X }`. Since the underlying type of z is Z, we can resolve `z.X` to mean `Option[Int]`, and then validate that `z `$<:$` T` and that `z` has a member `def foo: Option[Int]`.

3.3.11 Union and Intersection Types

```
UnionType         ::=  Type ‘|‘ Type
IntersectionType  ::=  Type ‘&‘ Type
```

Syntactically, the types `S | T` and `S & T` are infix types, where the infix operators are `|` and `&`, respectively (see infix types).

However, in this specification, $S \mid T$ and $S \& T$ refer to the underlying core concepts of *union and intersection types*, respectively.

- The type $S \mid T$ represents the set of values that are represented by *either* S or T.
- The type $S \& T$ represents the set of values that are represented by *both* S and T.

From the conformance rules rules on union and intersection types, we can show that $\&$ and $\mid$ are *commutative* and *associative*. Moreover, `&` is distributive over `|`. For any type A, B and C, all of the following relationships hold:

- $A \& B =:= B \& A$,
- $A \mid B =:= B \mid A$,
- $(A \& B) \& C =:= A \& (B \& C)$,
- $(A \mid B) \mid C =:= A \mid (B \mid C)$, and
- $A \& (B \mid C) =:= (A \& B) \mid (A \& C)$.

If C is a co- or contravariant type constructor, $C[A] \& C[B]$ can be simplified using the following rules:

- If C is covariant, $C[A] \& C[B] =:= C[A \& B]$
- If C is contravariant, $C[A] \& C[B] =:= C[A | B]$

The right-to-left validity of the above two rules can be derived from the definition of covariance and contravariance and the conformance rules of union and intersection types:

- When C is covariant, we can derive $C[A \& B] <: C[A] \& C[B]$.
- When C is contravariant, we can derive $C[A \mid B] <: C[A] \& C[B]$.

Join of a union type

In some situations, a union type might need to be widened to a non-union type. For this purpose, we define the *join* of a union type $T_1 \mid \ldots \mid T_n$ as the smallest intersection type of base class instances of $T_1, \ldots, T_n$. Note that union types might still appear as type arguments in the resulting type, this guarantees that the join is always finite.

For example, given

```
trait C[+T]
trait D
trait E
class A extends C[A] with D
class B extends C[B] with D with E
```

The join of $A \mid B$ is $C[A \mid B] \& D$

3.3.12 Skolem Types

```
SkolemType  ::=  ‘∃‘ skolemid ‘:‘ Type
```

Skolem types cannot directly be written in the concrete syntax. Moreover, although they are proper types, they can never be inferred to be part of the types of term definitions (`val`s, `var`s and `def`s).

They are exclusively used temporarily during subtyping derivations.

Skolem types are stable types. A skolem type of the form $\exists \alpha : T$ represents a stable reference to unknown value of type T. The identifier α is chosen uniquely every time a skolem type is created. However, as a skolem type is stable, it can be substituted in several occurrences in other types. When "copied" through substitution, all the copies retain the same α, and are therefore equivalent.

3.4 Methodic Types

```
TypeOrMethodic    ::=  Type
                    |  MethodicType
MethodicType      ::=  MethodType
                    |  PolyType
```

Methodic types are not real types. They are not part of the type lattice.

However, they share some meta-properties with types. In particular, when contained within other types that undertake some substitution, the substitution carries to the types within methodic types. It is therefore often convenient to think about them as types themselves.

Methodic types are used as the "declared type" of `def` definitions that have at least one term or type parameter list.

3.4.1 Method Types

```
MethodType        ::=  ‘(‘ MethodTypeParams ‘)‘ TypeOrMethodic
MethodTypeParams  ::=  ε
                    |  MethodTypeParam {‘,‘ MethodTypeParam}
MethodTypeParam   ::=  id ‘:‘ Type
```

A *method type* is denoted internally as $(Ps)U$, where (Ps) is a sequence of parameter names and types $(p_1 : T_1, ..., p_n : T_n)$ for some $n \geq 0$ and U is a (value or method) type. This type represents named methods that take arguments named $p_1, ..., p_n$ of types $T_1, ..., T_n$ and that return a result of type U.

Method types associate to the right: $(Ps_1)(Ps_2)U$ is treated as $(Ps_1)((Ps_2)U)$.

Method types do not exist as types of values. If a method name is used as a value, its type is implicitly converted to a corresponding function type.

Example

The declarations

```
def a: Int
def b (x: Int): Boolean
def c (x: Int) (y: String, z: String): String
```

produce the typings

```
a: Int
b: (Int) Boolean
c: (Int) (String, String) String
```

3.4.2 Polymorphic Method Types

```
PolyType          ::=  ‘[‘ PolyTypeParams ‘]‘ TypeOrMethodic
PolyTypeParams    ::=  PolyTypeParam {‘,‘ PolyTypeParam}
PolyTypeParam     ::=  ‘id‘ TypeBounds
```

A polymorphic method type, or *poly type* for short, is denoted internally as `[`*tps*`]`T where `[`*tps*`]` is a type parameter section `[`a_1 `>:` L_1 `<:` $U_1, \ldots, a_n$ `>:` L_n `<:` U_n`]` for some $n \geq 0$ and T is a (value or method) type. This type represents named methods that take type arguments $S_1, \ldots, S_n$ which conform to the lower bounds $L_1, \ldots, L_n$ and the upper bounds $U_1, \ldots, U_n$ and that yield results of type T.

Example

The declarations

```
def empty[A]: List[A]
def union[A <: Comparable[A]] (x: Set[A], xs: Set[A]): Set[A]
```

produce the typings

```
empty : [A >: Nothing <: Any] List[A]
union : [A >: Nothing <: Comparable[A]] (x: Set[A], xs: Set[A]) Set[A]
```

3.5 Operations on Types

This section defines a few meta-functions on types and methodic types.

- `baseType(`T`, `C`)`: computes the smallest type U of the form p`.`C`[`$T_1, \ldots, T_n$`]` such that $T <: U$.
- `asSeenFrom(`T`, `C`, `p`)`: rebases the type T visible inside the class C "as seen from" the prefix p.

- `memberType(`T`, `id`)`: finds a member of a type (`T.id`) and computes its underlying type or bounds.

These meta-functions are mutually recursive.

3.5.1 Base Type

The meta-function `baseType(`T`, `C`)`, where T is a proper type and C is a class identifier, computes the smallest type U of the form $p.C$ or $p.C$`[`$U_1, ..., U_n$`]` such that $T <: U$. If no such type exists, the function is not defined. The main purpose of `baseType` is to substitute prefixes and class type parameters along the inheritance chain.

We define `baseType(`T`, `C`)` as follows. For brevity, we write $p.X$`[`$U_1, ..., U_n$`]` instead of $p.X$ with $n = 0$.

- `baseType(`$T = p.C$`[`$T_1, ..., T_n$`], `C`)` $\triangleq T$
- `baseType(`$p.D$`[`$T_1, ..., T_n$`], `C`)` with $D \neq C \triangleq \sigma W$ if Q is defined where
  - D is declared as $D[\pm a_1 >: L_1 <: H_1, ..., \pm a_n >: L_n <: H_n]$ `extends` $P_1, ..., P_m$
  - $Q =$ `meet(baseType(`P_i`, `C`)` for all i such that `baseType(`P_i`, `C`)` is defined`)`
  - $W = Q$ if $p = \epsilon$ or if p is a package ref; otherwise, $W =$ `asSeenFrom(`Q`, `D`, `p`)` (in that case, p is a stable type and D must be declared inside another class B)
  - $\sigma = [a_1 := T_1, ..., a_n := T_n]$ the substitution of the declared type parameters of D by the actual type arguments
- `baseType(`$T_1 \& T_2$`, `C`)` $\triangleq$ `meet(baseType(`T_1`, `C`), baseType(`T_2`, `C`))`
- `baseType(`$T_1 \mid T_2$`, `C`)` $\triangleq$ `join(baseType(`T_1`, `C`), baseType(`T_2`, `C`))`
- `baseType(`T`, `C`)` $\triangleq$ `baseType(superType(`T`), `C`)` if `superType(`T`)` is defined

The definition above uses the following helper functions.

`superType(`T`)` computes the "next upper bound" of T, if it exists:

- `superType(`T`)` where T is a stable type is its underlying type
- `superType(`$p.X$`)` where $p.X$ is a non-class type designator is the upper bound of its underlying type definition
- `superType(`$([a_1 >: L_1 <: H_1, ..., a_n >: L_n <: H_n]$ `=>>` $U)[T_1, ..., T_n]$`)` is $[a_1 =: T_1, ..., a_n := T_n]U$ (i.e., the beta-reduction of the type lambda redex)

(i.e., the beta-reduction of the type lambda redex)

- superType($T[T_1, ..., T_n]$) is superType(T)$[T_1, ..., T_n]$ if superType(T) is defined

Note that the cases of superType do not overlap with each other nor with any baseType case other than the superType-based one. The cases of baseType therefore do not overlap with each other either. That makes baseType an algorithmic partial function.

meet($p.C[T_1, ..., T_n]$, $q.C[U_1, ..., U_n]$) computes an intersection of two (parameterized) class types for the same class, and join computes a union:

- if $p =:= q$ is false, then it is not defined
- otherwise, let W_i for $i \in 1, ..., n$ be:
  - $T_i \& U_i$ for meet (resp. $T_i \mid U_i$ for join) if the ith type parameter of C is covariant
  - $T_i \mid U_i$ for meet (resp. $T_i \& U_i$ for join) if the ith type parameter of C is contravariant
  - T_i if $T_i =:= U_i$ and the ith type parameter of C is invariant
  - not defined otherwise
- if any of the W_i are not defined, the result is not defined
- otherwise, the result is $p.C[W_1, ..., W_n]$

We generalize meet($T_1, ..., T_n$) for a sequence as:

- not defined for $n = 0$
- T_1 if $n = 1$
- meet(meet($T_1, ..., T_{n-1}$), T_n) if meet($T_1, ..., T_{n-1}$) is defined
- not defined otherwise

Examples

Given the following definitions:

```
trait Iterable[+A]
trait List[+A] extends Iterable[A]
trait Map[K, +V] extends Iterable[(K, V)]
trait Foo
```

we have the following baseType results:

```
baseType(List[Int], List) = List[Int]
```

- `baseType(List[Int], List) = List[Int]`
- `baseType(List[Int], Iterable) = Iterable[Int]`
- `baseType(List[A] & Iterable[B], Iterable) = meet(Iterable[A], Iterable[B]) = Iterable[A`
- `baseType(List[A] & Foo, Iterable) = Iterable[A]` (because `baseType(Foo, Iterable)` is not defined)
- `baseType(Int, Iterable)` is not defined
- `baseType(Map[Int, String], Iterable) = Iterable[(Int, String)]`
- `baseType(Map[Int, String] & Map[String, String], Map)` is not defined (because `K` is invariant)

3.5.2 As Seen From

The meta-function asSeenFrom(T, C, p), where T is a type or methodic type visible inside the class C and p is a stable type, rebases the type T "as seen from" the prefix p. Essentially, it substitutes this-types and class type parameters in T to appropriate types visible from outside. Since `T` is visible inside C, it can contain this-types and class type parameters of C itself as well as of all its enclosing classes. This-types of enclosing classes must be mapped to appropriate subprefixes of p, while class type parameters must be mapped to appropriate concrete type arguments.

asSeenFrom(T, C, p) only makes sense if p has a base type for C, i.e., if baseType(p, C) is defined.

We define asSeenFrom(T, C, p) where baseType(p, C) = $q.C[U_1, ..., U_n]$ as follows:

- If T is a reference to the ith class type parameter of some class D:
    - If baseType(p, D) $= r.D[W_1, ..., W_m]$ is defined, then W_i
    - Otherwise, if $q = \epsilon$ or q is a package ref, then T
    - Otherwise, q is a type, C must be defined in another class B and baseType(q, B) must be defined, then asSeenFrom(T, B, q)
- Otherwise, if T is a this-type D`.this`:
    - If D is a subclass of C and baseType(p, D) is defined, then p (this is always the case when $D = C$)
    - Otherwise, if $q = \epsilon$ or q is a package ref, then T
    - Otherwise, q is a type, C must be defined in another class B and baseType(q, B) must be

defined, then asSeenFrom(T', B, q)

- Otherwise, T where each if of its type components T_i is mapped to asSeenFrom(T_i, C, p).

For convenience, we generalize asSeenFrom to *type definitions* D.

- If D is an alias $= U$, then asSeenFrom(D, C, p) = asSeenFrom(U, C, p).
- If D is an abstract type definition with bounds $>: L <: H$, then asSeenFrom(D, C, p) = $>:$ asSeenFrom(L, C, p) $<:$ asSeenFrom(H, C, p).

3.5.3 Member Type

The meta-function memberType(T, id, p), where T is a proper type, id is a term or type identifier, and p is a stable type, finds a member of a type (`T.id`) and computes its underlying type (for a term) or type definition (for a type) as seen from the prefix p. For a term, it also computes whether the term is *stable*. memberType is the fundamental operation that computes the *underlying type* or *underlying type definition* of a named designator type.

The result M of a memberType is one of:

- undefined,
- a term result with underlying type or methodic type U and a *stable* flag,
- a class result with class C, or
- a type result with underlying type definition D.

As short-hand, we define memberType(T, id) to be the same as memberType(T, id, T) when T is a stable type.

We define memberType(T, id, p) as follows:

- If T is a possibly parameterized class type of the form $q.C[T_1, ..., T_n]$ (with $n \geq 0$):
    - Let m be the class member of C with name id.
    - If m is not defined, the result is undefined.
    - If m is a class declaration, the result is a class result with class m.
    - If m is a term definition in class D with declared type U, the result is a term result with underlying type asSeenFrom(U, D, p) and stable flag true if and only if m is stable.
    - If m is a type member definition in class D with declared type definition U, the result is a type result with underlying type definition asSeenFrom(U, D, p).

- If T is another monomorphic type designator of the form $q.X$:
  - Let U be `memberType(`q`, `X`)`
  - Let H be the upper bound of U
  - The result is `memberType(`H`, `id`, `p`)`
- If T is another parameterized type designator of the form $q.X[T_1, ..., T_n]$ (with $n \geq 0$):
  - Let U be `memberType(`q`, `X`)`
  - Let H be the upper bound of U
  - The result is `memberType(`$H[T_1, ..., T_n]$`, `id`, `p`)`
- If T is a parameterized type lambda of the form $([\pm a_1 >: L_1 <: H_1, ..., \pm a_n >: L_n <: H_n]$ `=>>` $U)[T_1, ..., T_n]$:
  - The result is `memberType(`$[a_1 := T_1, ..., a_n := T_n]U$`, `$id$`, `$p$`)`, i.e., we beta-reduce the type redex.
- If T is a refined type of the form T_1 `{` R `}`:
  - Let M_1 be the result of `memberType(`T_1`, `id`, `p`)`.
  - If the name of the refinement R is not id, let M_2 be undefined.
  - Otherwise, let M_2 be the type or type definition of the refinement R, as well as whether it is stable.
  - The result is `mergeMemberType(`M_1`, `M_2`)`.
- If T is a union type of the form $T_1 \mid T_2$:
  - Let J be the join of T.
  - The result is `memberType(`J`, `id`, `p`)`.
- If T is an intersection type of the form $T_1 \& T_2$:
  - Let M_1 be the result of `memberType(`T_1`, `id`, `p`)`.
  - Let M_2 be the result of `memberType(`T_2`, `id`, `p`)`.
  - The result is `mergeMemberType(`M_1`, `M_2`)`.
- If T is a recursive type of the form `{` α `=>` T_1 `}`:
  - The result is `memberType(`T_1`, `id`, `p`)`.

- If T is a stable type:
  - Let U be the underlying type of T.
  - The result is `memberType(`U`, `id`, `p`)`.
- Otherwise, the result is undefined.

We define the helper function `mergeMemberType(`M_1`, `M_2`)` as:

- If either M_1 or M_2 is undefined, the result is the other one.
- Otherwise, if either M_1 or M_2 is a class result, the result is that one.
- Otherwise, M_1 and M_2 must either both be term results or both be type results.
  - If they are term results with underlying types U_1 and U_2 and stable flags s_1 and s_2, the result is a term result whose underlying type is `meet(`U_1`, `U_2`)` and whose stable flag is $s_1 \vee s_2$.
  - If they are type results with underlying type definitions D_1 and D_2, the result is a type result whose underlying type definition is `intersect(`D_1`, `D_2`)`.

3.6 Relations between types

We define the following relations between types.

Name	Symbolically	Interpretation
Conformance	$T <: U$	Type T conforms to ("is a subtype of") type U.
Equivalence	$T =:= U$	T and U conform to each other.
Weak Conformance	$T <:_w U$	Augments conformance for primitive numeric types.
Compatibility		Type T conforms to type U after conversions.

3.6.1 Conformance

The conformance relation $(<:)$ is the smallest relation such that $S <: T$ is true if any of the following conditions hold. Note that the conditions are not all mutually exclusive.

- $S = T$ (i.e., conformance is reflexive by definition).
- S is `Nothing`.
- T is `AnyKind`.
- S is a stable type with underlying type S_1 and $S_1 <: T$.

- $S = p.x$ and $T = q.x$ are term designators and
  - `isSubPrefix(`p`, `q`)`.
- $S = p.X[S_1, ..., S_n]$ and $T = q.X[T_1, ..., T_n]$ are possibly parameterized type designators with $n \geq 0$ and:
  - `isSubPrefix(`p`, `q`)`, and
  - it is not the case that $p.x$ and $q.X$ are class type designators for different classes, and
  - for each $i \in 1, ..., n$:
    - the ith type parameter of $q.X$ is covariant and $S_i <: T_i$ [3], or
    - the ith type parameter of $q.X$ is contravariant and $T_i <: S_i$ [3], or
    - the ith type parameter of $q.X$ is invariant and:
      - S_i and T_i are types and $S_i =:= T_i$, or
      - S_i is a type and T_i is a wildcard type argument of the form $? >: L_2 <: H_2$ and $L_2 <: S_i$ and $S_i <: H_2$, or
      - S_i is a wildcard type argument of the form $? >: L_1 <: H_1$ and T_i is a wildcard type argument of the form $? >: L_2 <: H_2$ and $L_2 <: L_1$ and $H_1 <: H_2$ (i.e., the S_i "interval" is contained in the T_i "interval").
- $T = q.C[T_1, ..., T_n]$ with $n \geq 0$ and `baseType(`S`, `C`)` is defined and `baseType(S, C) $<: T$.
- $S = p.X[S_1, ..., S_n]$ and $p.X$ is non-class type designator and $H <: T$ where H is the upper bound of the underlying type definition of $p.X$.
- $S = p.C$ and $T = C$`.this` and C is the hidden class of an `object` and:
  - $p = \epsilon$ or p is a package ref, or
  - `isSubPrefix(`p`, `D`.this)` where D is the enclosing class of C.
- $S = C$`.this` and $T = q.C$ and C is the hidden class of an `object` and:
  - either $q = \epsilon$ or q is a package ref, or
  - `isSubPrefix(`D`.this, `q`)` where D is the enclosing class of C.
- $S = S_1 \mid S_2$ and $S_1 <: T$ and $S_2 <: T$.
- $T = T_1 \mid T_2$ and either $S <: T_1$ or $S <: T_2$.

- $T = T_1 \& T_2$ and $S <: T_1$ and $S <: T_2$.
- $S = S_1 \& S_2$ and either $S_1 <: T$ or $S_2 <: T$.
- $S = S_1$ `@a` and $S_1 <: T$.
- $T = T_1$ `@a` and $S <: T_1$ (i.e., annotations can be dropped).
- $T = q.X$ and $q.X$ is a non-class type designator and $S <: L$ where L is the lower bound of the underlying type definition of $q.X$.
- $S = p.X$ and $p.X$ is a non-class type designator and $H <: T$ where H is the upper bound of the underlying type definition of $p.X$.
- $S = [\pm a_1 >: L_1 <: H_1, ..., \pm a_n >: L_n <: H_n]$ `=>>` S_1 and $T = [\pm b_1 >: M_1 <: G_1, ..., \pm b_n >: M_n <: G_n]$ `=>>` T_1, and given $\sigma = [b_1 := a_1, ..., b_n := a_n]$:
  - $S_1 <: \sigma T_1$, and
  - for each $i \in 1, ..., n$:
    - the variance of a_i conforms to the variance of b_i ($+$ conforms to $+$ and ϵ, $-$ conforms to $-$ and ϵ, and ϵ conforms to ϵ), and
    - $\sigma(>: M_i <: G_i)$ is contained in $>: L_i <: H_i$ (i.e., $L_i <: \sigma M_i$ and $\sigma G_i <: H_i$).
- $S = p.X$ and $T = [\pm b_1 >: M_1 <: G_1, ..., \pm b_n >: M_n <: G_n]$ `=>>` T_1 and S is a type constructor with n type parameters and:
  - $([\pm a_1 >: L_1 <: H_1, ..., \pm a_n >: L_n <: H_n]$ `=>>` $S[a_1, ..., a_n]) <: T$ where the a_i are copies of the type parameters of S (i.e., we can eta-expand S to compare it to a type lambda).
- $T = T_1$ `{` R `}` and $S <: T_1$ and, given $p = S$ if S is a stable type and $p = \exists\alpha : S$ otherwise:
  - $R =$ `type` $X >: L <: H$ and `memberType(`p`,` X`)` is a class result for C and $L <: p.C$ and $p.C <: H$, or
  - $R =$ `type` $X >: L_2 <: H_2$ and `memberType(`p`,` X`)` is a type result with bounds $>: L_1 <: H_1$ and $L_2 <: L_1$ and $H_1 <: H_2$, or
  - $R =$ `val` $X : T_2$ and `memberType(`p`,` X`)` is a stable term result with type S_2 and $S_2 <: T_2$, or
  - $R =$ `def` $X : T_2$ and `memberType(`p`,` X`)` is a term result with type S_2 and T_2 is a type and $S_2 <: T_2$, or

$S_2 <: T_2$, or

  - $R =$ `def` $X : T_2$ and `memberType(`p`, `X`)` is a term result with methodic type S_2 and T_2 is a methodic type and `matches(`S_2`, `T_2`)`.

- $S = S_1$ `{` R `}` and $S_1 <: T$.
- $S =$ `{` α `=>` S_1 `}` and $T =$ `{` β `=>` T_1 `}` and $S_1 <: [\beta := \alpha]T_1$.
- $T =$ `{` β `=>` T_1 `}` and S is a proper type but not a recursive type and $p' <: [\beta := p]T_1$ where:
  - p is S if S is a stable type and $\exists\alpha : S$ otherwise, and
  - p' is the result of replacing any top-level recursive type `{` γ `=>` Z `}` in p with $[\gamma := p]Z$ (TODO specify this better).
- $S =$ (`=>` S_1) and $T =$ (`=>` T_1) and $S_1 <: T_1$.
- $S =$ `scala.Null` and:
  - $T = q.C[T_1, ..., T_n]$ with $n \geq 0$ and C does not derive from `scala.AnyVal` and C is not the hidden class of an `object`, or
  - $T = q.x$ is a term designator with underlying type U and `scala.Null` $<: U$, or
  - $T = T_1$ `{` R `}` and `scala.Null` $<: T_1$, or
  - $T =$ `{` β `=>` T_1 `}` and `scala.Null` $<: T_1$.
- S is a stable type and $T = q.x$ is a term designator with underlying type T_1 and T_1 is a stable type and $S <: T_1$.
- $S = S_1$ `{` R `}` and $S_1 <: T$.
- $S =$ `{` α `=>` S_1 `}` and $S_1 <: T$.

We define `isSubPrefix(`p`, `q`)` where p and q are prefixes as:

- If both p and q are types, then $p <: q$.
- Otherwise, $p = q$ (for empty prefixes and package refs).

We define `matches(`S`, `T`)` where S and T are types or methodic types as:

- If S and T are types, then $S <: T$.
- If S and T are method types $(a_1 : S_1, ..., a_n : S_n)S'$ and $(b_1 : T_1, ..., b_n : T_n)T'$, then $\sigma S_i =:= T_i$ for each i and `matches(`$\sigma S'$`, `T'`)`, where $\sigma = [a_1 := b_1, ..., a_n := b_n]$.

- If S and T are poly types $[a_1 >: L_{s1} <: H_{s1}, \ldots, a_n >: L_{sn} <: H_{sn}]S'$ and $[b_1 >: L_{t1} <: H_{t1}, \ldots, b_n >: L_{tn} <: H_{tn}]T'$, then $\sigma L_{si} =:= L_{ti}$ and $\sigma H_{si} =:= H_{ti}$ for each i and `matches(`$\sigma S'$`, `T'`)`, where $\sigma = [a_1 := b_1, \ldots, a_n := b_n]$.

Note that conformance in Scala is *not* transitive. Given two abstract types A and B, and one abstract `type` $C >: A <: B$ available on prefix p, we have $A <: p.C$ and $C <: p.B$ but not necessarily $A <: B$.

Least upper bounds and greatest lower bounds

The $(<:)$ relation forms pre-order between types, i.e. it is transitive and reflexive. This allows us to define *least upper bounds* and *greatest lower bounds* of a set of types in terms of that order.

- the *least upper bound* of `A` and `B` is the smallest type `L` such that `A` <: `L` and `B` <: `L`.
- the *greatest lower bound* of `A` and `B` is the largest type `G` such that `G` <: `A` and `G` <: `B`.

By construction, for all types `A` and `B`, the least upper bound of `A` and `B` is `A | B`, and their greatest lower bound is `A & B`.

3.6.2 Equivalence

Equivalence is defined as mutual conformance.

$S =:= T$ if and only if both $S <: T$ and $T <: S$.

3.6.3 Weak Conformance

In some situations Scala uses a more general conformance relation. A type S *weakly conforms* to a type T, written $S <:_w T$, if $S <: T$ or both S and T are primitive number types and S precedes T in the following ordering.

$$\begin{array}{lll}
\textbf{Byte} & <:_w & \textbf{Short} \\
\textbf{Short} & <:_w & \textbf{Int} \\
\textbf{Char} & <:_w & \textbf{Int} \\
\textbf{Int} & <:_w & \textbf{Long} \\
\textbf{Long} & <:_w & \textbf{Float} \\
\textbf{Float} & <:_w & \textbf{Double}
\end{array}$$

A *weak least upper bound* is a least upper bound with respect to weak conformance.

3.6.4 Compatibility

A type T is *compatible* to a type U if T (or its corresponding function type) weakly conforms to U after applying eta-expansion. If T is a method type, it's converted to the corresponding function type. If the types do not weakly conform, the following alternatives are checked in order:

type. If the types do not weakly conform, the following alternatives are checked in order:

- dropping by-name modifiers: if U is of the shape $=> U'$ (and T is not), $T <:_w U'$;
- SAM conversion: if T corresponds to a function type, and U declares a single abstract method whose type corresponds to the function type U', $T <:_w U'$.
- implicit conversion: there's an implicit conversion from T to U in scope;

Examples

Function compatibility via SAM conversion

Given the definitions

```
def foo(x: Int => String): Unit
def foo(x: ToString): Unit

trait ToString { def convert(x: Int): String }
```

The application `foo((x: Int) => x.toString)` resolves to the first overload, as it's more specific:

- `Int => String` is compatible to `ToString` -- when expecting a value of type `ToString`, you may pass a function literal from `Int` to `String`, as it will be SAM-converted to said function;
- `ToString` is not compatible to `Int => String` -- when expecting a function from `Int` to `String`, you may not pass a `ToString`.

3.7 Realizability

A type T is *realizable* if and only if it is inhabited by non-null values. It is defined as:

- A term designator $p.x$ with underlying type U is realizable if p is ϵ or a package ref or a realizable type and
  - `memberType(`p`, `x`)` has the stable flag, or
  - the type returned by `memberType(`p`, `x`)` is realizable.
- A stable type that is not a term designator is realizable.
- Another type T is realizable if
  - T is concrete, and
  - T has good bounds.

A concrete type T has good bounds if all of the following apply:

- all its non-class type members have good bounds, i.e., their bounds L and H are such that $L <: H$,
- all its type refinements have good bounds, and
- for all base classes C of T:
  - `baseType(`T`, `C`)` is defined with some result $p.C[T_1, ..., T_n]$, and
  - for all $i \in 1, ..., n$, T_i is a real type or (when it is a wildcard type argument) it has good bounds.

Note: it is possible for `baseType(`T`, `C`)` not to be defined because of the `meet` computation, which may fail to merge prefixes and/or invariant type arguments.

3.8 Type Erasure

A type is called *generic* if it contains type arguments or type variables. *Type erasure* is a mapping from (possibly generic) types to non-generic types. We write $|T|$ for the erasure of type T. The erasure mapping is defined as follows.

- The erasure of `AnyKind` is `Object`.
- The erasure of a non-class type designator is the erasure of its underlying upper bound.
- The erasure of a term designator is the erasure of its underlying type.
- The erasure of the parameterized type `scala.Array`$[T_1]$ is `scala.Array`$[|T_1|]$.
- The erasure of every other parameterized type $T[T_1, ..., T_n]$ is $|T|$.
- The erasure of a stable type p is the erasure of the underlying type of p.
- The erasure of a by-name type `=> `T_1 is `scala.Function0`.
- The erasure of an annotated type $T_1 a$ is $|T_1|$.
- The erasure of a refined type T_1 `{ `R` }` is $|T_1|$.
- The erasure of a recursive type `{ `α` => `T_1` }` and the associated recursive this type α is $|T_1|$.
- The erasure of a union type $S \mid T$ is the *erased least upper bound* (*elub*) of the erasures of S and T.
- The erasure of an intersection type $S \& T$ is the *eglb* (erased greatest lower bound) of the erasures of S and T.

The erased LUB is computed as follows:

- if both argument are arrays of objects, an array of the erased LUB of the element types
- if both arguments are arrays of same primitives, an array of this primitive
- if one argument is array of primitives and the other is array of objects, `Object`
- if one argument is an array, `Object`
- otherwise a common superclass or trait S of the argument classes, with the following two properties:
  - S is minimal: no other common superclass or trait derives from S, and
  - S is last: in the linearization of the first argument type $|A|$ there are no minimal common superclasses or traits that come after S. The reason to pick last is that we prefer classes over traits that way, which leads to more predictable bytecode and (?) faster dynamic dispatch.

The rules for $eglb(A, B)$ are given below in pseudocode:

```
eglb(scala.Array[A], JArray[B]) = scala.Array[eglb(A, B)]
eglb(scala.Array[T], _)         = scala.Array[T]
eglb(_, scala.Array[T])         = scala.Array[T]
eglb(A, B)                      = A                     if A extends B
eglb(A, B)                      = B                     if B extends A
eglb(A, _)                      = A                     if A is not a
trait
eglb(_, B)                      = B                     if B is not a
trait
eglb(A, _)                      = A                     // use first
```

1. In the literature, this is often achieved through De Bruijn indices or through alpha-renaming when needed. In a concrete implementation, this is often achieved through retaining *symbolic* references in a symbol table. ↩
2. A reference to a structurally defined member (method call or access to a value or variable) may generate binary code that is significantly slower than an equivalent code to a non-structural member. ↩
3. In these cases, if `T_i` and/or `U_i` are wildcard type arguments, the simplification rules for parameterized types allow to reduce them to real types. ↩