

# Types

## Types

```
Type          ::= FunctionArgTypes '>' Type
                | InfixType [ExistentialClause]
FunctionArgTypes ::= InfixType
                | '(' [ ParamType {',' ParamType } ] ')'
ExistentialClause ::= 'forSome' '{' ExistentialDcl
                    {semi ExistentialDcl} '}'
ExistentialDcl   ::= 'type' TypeDcl
                | 'val' ValDcl
InfixType        ::= CompoundType {id [nl] CompoundType}
CompoundType     ::= AnnotType {'with' AnnotType} [Refinement]
                | Refinement
AnnotType        ::= SimpleType {Annotation}
SimpleType       ::= SimpleType TypeArgs
                | SimpleType '#' id
                | StableId
                | Path '.' 'type'
                | Literal
                | '(' Types ')'
TypeArgs         ::= '[' Types ']'
Types            ::= Type {',' Type}
```

We distinguish between first-order types and type constructors, which take type parameters and yield types. A subset of first-order types called *value types* represents sets of (first-class) values. Value types are either *concrete* or *abstract*.

Every concrete value type can be represented as a *class type*, i.e. a type designator that refers to a class or a trait <sup>1</sup>, or as a compound type representing an intersection of types, possibly with a refinement that further constrains the types of its members. Abstract value types are introduced by type parameters and abstract type bindings. Parentheses in types can be used for grouping.

Non-value types capture properties of identifiers that are not values. For example, a type constructor does not directly specify a type of values. However,

---

<sup>1</sup>We assume that objects and packages also implicitly define a class (of the same name as the object or package, but inaccessible to user programs).

when a type constructor is applied to the correct type arguments, it yields a first-order type, which may be a value type.

Non-value types are expressed indirectly in Scala. E.g., a method type is described by writing down a method signature, which in itself is not a real type, although it gives rise to a corresponding method type. Type constructors are another example, as one can write `type Swap[m[_], _], a,b] = m[b, a]`, but there is no syntax to write the corresponding anonymous type function directly.

## Paths

```
Path      ::= StableId
            | [id '.'] this
StableId   ::= id
            | Path '.' id
            | [id '.'] 'super' [ClassQualifier] '.' id
ClassQualifier ::= '[' id ']'
```

Paths are not types themselves, but they can be a part of named types and in that function form a central role in Scala's type system.

A path is one of the following.

- The empty path (which cannot be written explicitly in user programs).
- `C.this`, where `C` references a class. The path `this` is taken as a shorthand for `C.this` where `C` is the name of the class directly enclosing the reference.
- `p.x` where `p` is a path and `x` is a stable member of `p`. *Stable members* are packages or members introduced by object definitions or by value definitions of non-volatile types.
- `C.super.x` or `C.super[M].x` where `C` references a class and `x` references a stable member of the super class or designated parent class `M` of `C`. The prefix `super` is taken as a shorthand for `C.super` where `C` is the name of the class directly enclosing the reference.

A *stable identifier* is a path which ends in an identifier.

## Value Types

Every value in Scala has a type which is of one of the following forms.

### Singleton Types

```
SimpleType ::= Path '.' 'type'
```

A *singleton type* is of the form `p.type`. Where `p` is a path pointing to a value which conforms to `scala.AnyRef`, the type denotes the set of values consisting of `null` and the value denoted by `p` (i.e., the value `v` for which `v eq p`). Where

the path does not conform to `scala.AnyRef` the type denotes the set consisting of only the value denoted by *p*.

## Literal Types

`SimpleType ::= Literal`

A literal type `lit` is a special kind of singleton type which 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 types for which there is dedicated syntax except `Unit`. This includes the numeric types (other than `Byte` and `Short` which don't currently have syntax), `Boolean`, `Char`, `String` and `Symbol`.

## Stable Types

A *stable type* is a singleton type, a literal type, or a type that is declared to be a subtype of trait `scala.Singleton`.

## Type Projection

`SimpleType ::= SimpleType '#' id`

A *type projection* `T#x` references the type member named *x* of type *T*.

## Type Designators

`SimpleType ::= StableId`

A *type designator* refers to a named value type. It can be simple or qualified. All such type designators are shorthands for type projections.

Specifically, the unqualified type name *t* where *t* is bound in some class, object, or package *C* is taken as a shorthand for `C.this.type#t`. If *t* is not bound in a class, object, or package, then *t* is taken as a shorthand for `.type#t`.

A qualified type designator has the form `p.t` where *p* is a path and *t* is a type name. Such a type designator is equivalent to the type projection `p.type#t`.

Example

Some type designators and their expansions are listed below. We assume a local type parameter *t*, a value `maintable` with a type member `Node` and the standard class `scala.Int`,

Designator	Expansion
t	.type#t
Int	scala.type#Int
scala.Int	scala.type#Int
data.maintable.Node	data.maintable.type#Node

## Parameterized Types

```
SimpleType      ::= SimpleType TypeArgs
TypeArgs        ::= '[' Types '['
```

A *parameterized type*  $T[T_1, \dots, T_n]$  consists of a type designator  $T$  and type parameters  $T_1, \dots, T_n$  where  $n \geq 1$ .  $T$  must refer to a type constructor which takes  $n$  type parameters  $a_1, \dots, a_n$ .

Say the type parameters have lower bounds  $L_1, \dots, L_n$  and upper bounds  $U_1, \dots, U_n$ . The parameterized type is well-formed if each actual type parameter *conforms to its bounds*, i.e.  $\sigma L_i <: T_i <: \sigma U_i$  where  $\sigma$  is the substitution  $[a_1 := T_1, \dots, a_n := T_n]$ .

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[_], X] { ... }
class S[K <: String] { ... }
class G[M[Z <: I], I] { ... }
```

the following parameterized types are well formed:

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

```
F[List, Int]
G[S, String]
```

Example

Given the above type definitions, the following types are ill-formed:

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

F[Int, Boolean]      // illegal: Int is not a type constructor
F[TreeMap, Int]      // illegal: TreeMap takes two parameters,
```

```

G[S, Int]           // F expects a constructor taking one
                    // illegal: S constrains its parameter to
                    // conform to String,
                    // G expects type constructor with a parameter
                    // that conforms to Int

```

## Tuple Types

```
SimpleType ::= '(' Types ')'
```

A *tuple type*  $(T_1, \dots, T_n)$  is an alias for the class `scala.Tuple` $n$ `[$T_1$, ... , $T_n$]`, where  $n \geq 2$ .

Tuple classes are case classes whose fields can be accessed using selectors `_1` , ... , `_n`. Their functionality is abstracted in a corresponding `Product` trait. The  $n$ -ary tuple class and product trait are defined at least as follows in the standard Scala library (they might also add other methods and implement other traits).

```

case class Tuple
```

 $n$ `[$T_1$, ... , $T_n$](_1: $T_1$, ... , _n: $T_n$)
extends Product` $n$ `[$T_1$, ... , $T_n$]

trait Product` $n$ `[$T_1$, ... , $T_n$] {
 override def productArity =`  $n$ `def _1: $T_1$
 ...
 def _n: $T_n$
}`

## Annotated Types

```
AnnotType ::= SimpleType {Annotation}
```

An *annotated type*  $T \ a_1, \dots, a_n$  attaches annotations  $a_1, \dots, a_n$  to the type  $T$ .

Example

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

```
String @suspendable
```

## Compound Types

```

CompoundType ::= AnnotType {'with' AnnotType} [Refinement]
               | Refinement
Refinement    ::= [nl] '{' RefineStat {semi RefineStat} '}'
RefineStat    ::= Dcl
               | 'type' TypeDef
               |

```

A *compound type*  $T_1$  **with** ... **with**  $T_n$

$R$  represents objects with members as given in the component types  $T_1, \dots, T_n$  and the refinement

$R$ . A refinement

$R$  contains declarations and type definitions. If a declaration or definition overrides a declaration or definition in one of the component types  $T_1, \dots, T_n$ , the usual rules for overriding apply; otherwise the declaration or definition is said to be “structural”<sup>2</sup>.

Within a method declaration in a structural refinement, the type of any value parameter may only refer to type parameters or abstract types that are contained inside the refinement. That is, it must refer either to a type parameter of the method itself, or to a type definition within the refinement. This restriction does not apply to the method’s result type.

If no refinement is given, the empty refinement is implicitly added, i.e.  $T_1$  **with** ... **with**  $T_n$  is a shorthand for  $T_1$  **with** ... **with**  $T_n$

.

A compound type may also consist of just a refinement

$R$  with no preceding component types. Such a type is equivalent to `AnyRef`

$R$ .

Example

The following example shows how to declare and use a method which has a parameter type that contains a refinement with structural declarations.

```
case class Bird (val name: String) extends Object {
    def fly(height: Int) = ...
...
}
case class Plane (val callsign: String) extends Object {
    def fly(height: Int) = ...
...
}
def takeoff(
    runway: Int,
    r: { val callsign: String; def fly(height: Int) }) = {
    tower.print(r.callsign + " requests take-off on runway " + runway)
    tower.read(r.callsign + " is clear for take-off")
    r.fly(1000)
}
val bird = new Bird("Polly the parrot"){ val callsign = name }
val a380 = new Plane("TZ-987")
```

---

<sup>2</sup>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.

```
takeoff(42, bird)
takeoff(89, a380)
```

Although `Bird` and `Plane` do not share any parent class other than `Object`, the parameter  $r$  of method `takeoff` is defined using a refinement with structural declarations to accept any object that declares a value `callsign` and a `fly` method.

## Infix Types

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

An *infix type*  $T_1 \text{ op } T_2$  consists of an infix operator `op` which gets applied to two type operands  $T_1$  and  $T_2$ . The type is equivalent to the type application `op[ $T_1, T_2$ ]`. The infix operator `op` may be an arbitrary identifier.

All type infix operators have the same precedence; parentheses have to be used for grouping. The associativity of a type operator is determined as for term operators: type operators ending in a colon `:` are right-associative; all other operators are left-associative.

In a sequence of consecutive type infix operations  $t_0 \text{ op } t_1 \text{ op } t_2 \dots \text{ op } t_n$ , all operators  $\text{op}_1, \dots, \text{op}_n$  must have the same associativity. If they are all left-associative, the sequence is interpreted as  $(\dots (t_0 \text{ op}_1 t_1) \text{ op}_2 \dots) \text{ op}_n t_n$ , otherwise it is interpreted as  $t_0 \text{ op}_1 (t_1 \text{ op}_2 (\dots \text{ op}_n t_n) \dots)$ .

## Function Types

```
Type          ::= FunctionArgs '=>' Type
FunctionArgs   ::= InfixType
                  | '(' [ ParamType {',' ParamType } ] ')'
```

The type  $(T_1, \dots, T_n) \Rightarrow U$  represents the set of function values that take arguments of types  $T_1, \dots, T_n$  and yield results of type  $U$ . In the case of exactly one argument type  $T \Rightarrow U$  is a shorthand for  $(T) \Rightarrow U$ . 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.  $S \Rightarrow T \Rightarrow U$  is the same as  $S \Rightarrow (T \Rightarrow U)$ .

Function types are shorthands for class types that define `apply` functions. Specifically, the  $n$ -ary function type  $(T_1, \dots, T_n) \Rightarrow U$  is a shorthand for the class type `Function$_n$[ $T_1$  , ... ,  $T_n$ ,  $U$ ]`. Such class types are defined in the Scala library for  $n$  between 0 and 22 as follows.

```
package scala
trait Function_n[-T1 , ... , -T$_n$, +R] {
  def apply(x1: T1 , ... , x$_n$: T$_n$): R
```

```

    override def toString = "<function>"
  }

```

Hence, function types are covariant in their result type and contravariant in their argument types.

## Existential Types

```

Type          ::= InfixType ExistentialClauses
ExistentialClauses ::= 'forSome' '{' ExistentialDcl
                    {semi ExistentialDcl} '}'
ExistentialDcl  ::= 'type' TypeDcl
                    | 'val' ValDcl

```

An *existential type* has the form  $\text{\$T\$ forSome } \{ \text{\$Q\$} \}$  where  $Q$  is a sequence of type declarations.

Let  $t_1[\textit{tps\_1}] >: L_1 <: U_1, \dots, t_n[\textit{tps\_n}] >: L_n <: U_n$  be the types declared in  $Q$  (any of the type parameter sections  $[\text{\$}\mathit{tps\_i}\text{\$}]$  might be missing). The scope of each type  $t_i$  includes the type  $T$  and the existential clause  $Q$ . The type variables  $t_i$  are said to be *bound* in the type  $\text{\$T\$ forSome } \{ \text{\$Q\$} \}$ . Type variables which occur in a type  $T$  but which are not bound in  $T$  are said to be *free* in  $T$ .

A *type instance* of  $\text{\$T\$ forSome } \{ \text{\$Q\$} \}$  is a type  $\sigma T$  where  $\sigma$  is a substitution over  $t_1, \dots, t_n$  such that, for each  $i$ ,  $\sigma L_i <: \sigma t_i <: \sigma U_i$ . The set of values denoted by the existential type  $\text{\$T\$ forSome } \{ \text{\$Q\$} \}$  is the union of the set of values of all its type instances.

A *skolemization* of  $\text{\$T\$ forSome } \{ \text{\$Q\$} \}$  is a type instance  $\sigma T$ , where  $\sigma$  is the substitution  $[t'_1/t_1, \dots, t'_n/t_n]$  and each  $t'_i$  is a fresh abstract type with lower bound  $\sigma L_i$  and upper bound  $\sigma U_i$ .

**Simplification Rules** Existential types obey the following four equivalences:

1. Multiple for-clauses in an existential type can be merged. E.g.,  $\text{\$T\$ forSome } \{ \text{\$Q\$} \} \text{\$ forSome } \{ \text{\$Q'\$} \}$  is equivalent to  $\text{\$T\$ forSome } \{ \text{\$Q\$ ; \$Q'\$} \}$ .
2. Unused quantifications can be dropped. E.g.,  $\text{\$T\$ forSome } \{ \text{\$Q\$ ; \$Q'\$} \}$  where none of the types defined in  $Q'$  are referred to by  $T$  or  $Q$ , is equivalent to  $\text{\$T\$ forSome } \{ \text{\$Q\$} \}$ .
3. An empty quantification can be dropped. E.g.,  $\text{\$T\$ forSome } \{ \}$  is equivalent to  $T$ .
4. An existential type  $\text{\$T\$ forSome } \{ \text{\$Q\$} \}$  where  $Q$  contains a clause  $\text{\$t[\mathit{tps}] >: L <: U\$}$  is equivalent to the type  $\text{\$T'\$ forSome } \{ \text{\$Q\$} \}$  where  $T'$  results from  $T$  by replacing every covariant occurrence of  $t$  in  $T$  by  $U$  and by replacing every contravariant occurrence of  $t$  in  $T$  by  $L$ .



**Existential Quantification over Values** As a syntactic convenience, the bindings clause in an existential type may also contain value declarations `val  $x$ :  $T$` . An existential type  `$T$  forSome {  $Q$ ; val  $x$ :  $S$ ;  $Q$  }` is treated as a shorthand for the type  `$T$ ' forSome {  $Q$ ; type  $t$  <:  $S$  with Singleton;  $Q$ ' }`, where  $t$  is a fresh type name and  $T'$  results from  $T$  by replacing every occurrence of  `$x$ .type` with  $t$ .

## Placeholder Syntax for Existential Types

`WildcardType ::= ' _ ' TypeBounds`

Scala supports a placeholder syntax for existential types. A *wildcard type* is of the form `_<math>\backslash; >: <math>\backslash; L, <math>\backslash; <: <math>\backslash; U</math>`. Both bound clauses may be omitted. If a lower bound clause `>:<math>\backslash; L</math>` is missing, `>:<math>\backslash; \text{scala.Nothing}` is assumed. If an upper bound clause `<:<math>\backslash; U</math>` is missing, `<:<math>\backslash; \text{scala.Any}` is assumed. A wildcard type is a shorthand for an existentially quantified type variable, where the existential quantification is implicit.

A wildcard type must appear as a type argument of a parameterized type. Let  `$T = p.c[targs, T, targs']$`  be a parameterized type where  `$targs, targs'$`  may be empty and  $T$  is a wildcard type `_<math>\backslash; >: <math>\backslash; L, <math>\backslash; <: <math>\backslash; U</math>`. Then  $T$  is equivalent to the existential type

`$p.c[\textit{\textbf{targs}}, t, \textit{\textbf{targs}}']$  forSome { type  $t$  >:  $L$  <:  $U$  }`

where  $t$  is some fresh type variable. Wildcard types may also appear as parts of infix types, function types, or tuple types. Their expansion is then the expansion in the equivalent parameterized type.

Example

Assume the class definitions

```
class Ref[T]
abstract class Outer { type T }
```

Here are some examples of existential types:

```
Ref[T] forSome { type T <: java.lang.Number }
Ref[x.T] forSome { val x: Outer }
Ref[x_type # T] forSome { type x_type <: Outer with Singleton }
```

The last two types in this list are equivalent. An alternative formulation of the first type above using wildcard syntax is:

```
Ref[_ <: java.lang.Number]
```

Example

The type `List[List[_]]` is equivalent to the existential type

```
List[List[t] forSome { type t }]
```

Example

Assume a covariant type

```
class List[+T]
```

The type

```
List[T] forSome { type T <: java.lang.Number }
```

is equivalent (by simplification rule 4 above) to

```
List[java.lang.Number] forSome { type T <: java.lang.Number }
```

which is in turn equivalent (by simplification rules 2 and 3 above) to  
`List[java.lang.Number]`.

## Non-Value Types

The types explained in the following do not denote sets of values, nor do they appear explicitly in programs. They are introduced in this report as the internal types of defined identifiers.

## Method Types

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

Method types associate to the right:  $(Ps\_1)(Ps\_2)U$  is treated as  $(Ps\_1)((Ps\_2)U)$ .

A special case are types of methods without any parameters. They are written here  $\Rightarrow T$ . Parameterless methods name expressions that are re-evaluated each time the parameterless method name is referenced.

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
```

## Polymorphic Method Types

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

## Type Constructors

A *type constructor* is represented internally much like a polymorphic method type.  $[\pm a_1 >: L_1 <: U_1, \dots, \pm a_n >: L_n <: U_n] T$  represents a type that is expected by a type constructor parameter or an abstract type constructor binding with the corresponding type parameter clause.

Example

Consider this fragment of the `Iterable[+X]` class:

```
trait Iterable[+X] {
  def flatMap[newType[+X] <: Iterable[X], S](f: X => newType[S]): newType[S]
}
```

Conceptually, the type constructor `Iterable` is a name for the anonymous type `[+X] Iterable[X]`, which may be passed to the `newType` type constructor parameter in `flatMap`.

## Base Types and Member Definitions

Types of class members depend on the way the members are referenced. Central here are three notions, namely: 1. the notion of the set of base types of a type  $T$ , 1. the notion of a type  $T$  in some class  $C$  seen from some prefix type  $S$ , 1. the notion of the set of member bindings of some type  $T$ .

These notions are defined mutually recursively as follows.

1. The set of *base types* of a type is a set of class types, given as follows.
  - The base types of a class type  $C$  with parents  $T_1, \dots, T_n$  are  $C$  itself, as well as the base types of the compound type  $\$T\_1\$ \text{ with } \dots \text{ with } \$T\_n\$ \{ \$R\$ \}$ .
  - The base types of an aliased type are the base types of its alias.
  - The base types of an abstract type are the base types of its upper bound.
  - The base types of a parameterized type  $\$C\$[\$T\_1\$, \dots, \$T\_n\$]$  are the base types of type  $C$ , where every occurrence of a type parameter  $a_i$  of  $C$  has been replaced by the corresponding parameter type  $T_i$ .
  - The base types of a singleton type  $\$p\$.type$  are the base types of the type of  $p$ .
  - The base types of a compound type  $\$T\_1\$ \text{ with } \$\dots\$ \text{ with } \$T\_n\$ \{ \$R\$ \}$  are the *reduced union* of the base classes of all  $T_i$ 's. This means: Let the multi-set  $\mathcal{S}$  be the multi-set-union of the base types of all  $T_i$ 's. If  $\mathcal{S}$  contains several type instances of the same class, say  $\$S^i\$[\$T^i\_1\$, \dots, \$T^i\_n\$]$  ( $i \in I$ ), then all those instances are replaced by one of them which conforms to all others. It is an error if no such instance exists. It follows that the reduced union, if it exists, produces a set of class types, where different types are instances of different classes.
  - The base types of a type selection  $\$S\$ \# \$T\$$  are determined as follows. If  $T$  is an alias or abstract type, the previous clauses apply. Otherwise,  $T$  must be a (possibly parameterized) class type, which is defined in some class  $B$ . Then the base types of  $\$S\$ \# \$T\$$  are the base types of  $T$  in  $B$  seen from the prefix type  $S$ .
  - The base types of an existential type  $\$T\$ \text{ forSome } \{ \$Q\$ \}$  are all types  $\$S\$ \text{ forSome } \{ \$Q\$ \}$  where  $S$  is a base type of  $T$ .
1. The notion of a type  $T$  in class  $C$  seen from some prefix type  $S$  makes sense only if the prefix type  $S$  has a type instance of class  $C$  as a base type, say  $\$S'\$[\$T\_1\$, \dots, \$T\_n\$]$ . Then we define as follows.
  - If  $\$S\$ = \$\epsilon\$.type$ , then  $T$  in  $C$  seen from  $S$  is  $T$  itself.
  - Otherwise, if  $S$  is an existential type  $\$S'\$ \text{ forSome } \{ \$Q\$ \}$ , and  $T$  in  $C$  seen from  $S'$  is  $T'$ , then  $T$  in  $C$  seen from  $S$  is  $\$T'\$ \text{ forSome } \{ \$\backslash, Q\backslash, \$\}$ .
  - Otherwise, if  $T$  is the  $i$ 'th type parameter of some class  $D$ , then
    - If  $S$  has a base type  $\$D\$[\$U\_1\$, \dots, \$U\_n\$]$ , for some type parameters  $[\$U\_1\$, \dots, \$U\_n\$]$ , then  $T$  in  $C$  seen from  $S$  is  $U_i$ .
    - Otherwise, if  $C$  is defined in a class  $C'$ , then  $T$  in  $C$  seen from  $S$  is the same as  $T$  in  $C'$  seen from  $S'$ .
    - Otherwise, if  $C$  is not defined in another class, then  $T$  in  $C$  seen from  $S$  is  $T$  itself.
  - Otherwise, if  $T$  is the singleton type  $\$D\$.this.type$  for some class  $D$  then
    - If  $D$  is a subclass of  $C$  and  $S$  has a type instance of class  $D$

- among its base types, then  $T$  in  $C$  seen from  $S$  is  $S$ .
- Otherwise, if  $C$  is defined in a class  $C'$ , then  $T$  in  $C$  seen from  $S$  is the same as  $T$  in  $C'$  seen from  $S'$ .
- Otherwise, if  $C$  is not defined in another class, then  $T$  in  $C$  seen from  $S$  is  $T$  itself.
- If  $T$  is some other type, then the described mapping is performed to all its type components.

If  $T$  is a possibly parameterized class type, where  $T$ 's class is defined in some other class  $D$ , and  $S$  is some prefix type, then we use “ $T$  seen from  $S$ ” as a shorthand for “ $T$  in  $D$  seen from  $S$ ”.

2. The *member bindings* of a type  $T$  are
  1. all bindings  $d$  such that there exists a type instance of some class  $C$  among the base types of  $T$  and there exists a definition or declaration  $d'$  in  $C$  such that  $d$  results from  $d'$  by replacing every type  $T'$  in  $d'$  by  $T'$  in  $C$  seen from  $T$ , and
  2. all bindings of the type's refinement, if it has one.

The *definition* of a type projection  $\mathbf{S\#T}$  is the member binding  $d_T$  of the type  $\mathbf{T}$  in  $\mathbf{S}$ . In that case, we also say that  $\mathbf{S\#T}$  is *defined by*  $d_T$ .

## Relations between types

We define the following relations between types.

Name	Symbolically	Interpretation
Equivalence	$T \equiv U$	$T$ and $U$ are interchangeable in all contexts.
Conformance	$T <: U$	Type $T$ conforms to (“is a subtype of”) type $U$ .
Weak Conformance	$T <:_w U$	Augments conformance for primitive numeric types.
Compatibility		Type $T$ conforms to type $U$ after conversions.

### Equivalence

Equivalence ( $\equiv$ ) between types is the smallest congruence<sup>3</sup> such that the following holds:

- If  $t$  is defined by a type alias `type  $\mathbf{t} = \mathbf{T}$` , then  $t$  is equivalent to  $T$ .
- If a path  $p$  has a singleton type  `$\mathbf{q}.\mathbf{type}$` , then  `$\mathbf{p}.\mathbf{type} \equiv \mathbf{q}.\mathbf{type}$` .

<sup>3</sup>A congruence is an equivalence relation which is closed under formation of contexts.

- If  $O$  is defined by an object definition, and  $p$  is a path consisting only of package or object selectors and ending in  $O$ , then  $O$.this.type \equiv p$.type$ .
- Two compound types are equivalent if the sequences of their component are pairwise equivalent, and occur in the same order, and their refinements are equivalent. Two refinements are equivalent if they bind the same names and the modifiers, types and bounds of every declared entity are equivalent in both refinements.
- Two method types are equivalent if:
  - neither are implicit, or they both are <sup>4</sup>;
  - they have equivalent result types;
  - they have the same number of parameters; and
  - corresponding parameters have equivalent types. Note that the names of parameters do not matter for method type equivalence.
- Two polymorphic method types are equivalent if they have the same number of type parameters, and, after renaming one set of type parameters by another, the result types as well as lower and upper bounds of corresponding type parameters are equivalent.
- Two existential types are equivalent if they have the same number of quantifiers, and, after renaming one list of type quantifiers by another, the quantified types as well as lower and upper bounds of corresponding quantifiers are equivalent.
- Two type constructors are equivalent if they have the same number of type parameters, and, after renaming one list of type parameters by another, the result types as well as variances, lower and upper bounds of corresponding type parameters are equivalent.

## Conformance

The conformance relation ( $<:$ ) is the smallest transitive relation that satisfies the following conditions.

- Conformance includes equivalence. If  $T \equiv U$  then  $T <: U$ .
- For every value type  $T$ , `scala.Nothing <: T <: scala.Any`.
- For every type constructor  $T$  (with any number of type parameters), `scala.Nothing <: T <: scala.Any`.
- For every value type  $T$ , `scala.Null <: T` unless `T <: scala.AnyVal`.
- A type variable or abstract type  $t$  conforms to its upper bound and its lower bound conforms to  $t$ .
- A class type or parameterized type conforms to any of its base-types.
- A singleton type `$p$.type` conforms to the type of the path  $p$ .
- A singleton type `$p$.type` conforms to the type `scala.Singleton`.
- A type projection `$T#$t$` conforms to `$U#$t$` if  $T$  conforms to  $U$ .

---

<sup>4</sup>A method type is implicit if the parameter section that defines it starts with the `implicit` keyword.

- A parameterized type  $T[T_1, \dots, T_n]$  conforms to  $T[U_1, \dots, U_n]$  if the following three conditions hold for  $i \in \{1, \dots, n\}$ :
  1. If the  $i$ 'th type parameter of  $T$  is declared covariant, then  $T_i <: U_i$ .
  2. If the  $i$ 'th type parameter of  $T$  is declared contravariant, then  $U_i <: T_i$ .
  3. If the  $i$ 'th type parameter of  $T$  is declared neither covariant nor contravariant, then  $U_i \equiv T_i$ .
- A compound type  $T_1$  with  $\dots$  with  $T_n$   $\{\mathcal{R}, \mathcal{Q}\}$  conforms to each of its component types  $T_i$ .
- If  $T <: U_i$  for  $i \in \{1, \dots, n\}$  and for every binding  $d$  of a type or value  $x$  in  $R$  there exists a member binding of  $x$  in  $T$  which subsumes  $d$ , then  $T$  conforms to the compound type  $U_1$  with  $\dots$  with  $U_n$   $\{\mathcal{R}, \mathcal{Q}\}$ .
- The existential type  $T$  forSome  $\{\mathcal{Q}, \mathcal{Q}, \mathcal{Q}\}$  conforms to  $U$  if its skolemization conforms to  $U$ .
- The type  $T$  conforms to the existential type  $U$  forSome  $\{\mathcal{Q}, \mathcal{Q}, \mathcal{Q}\}$  if  $T$  conforms to one of the type instances of  $U$  forSome  $\{\mathcal{Q}, \mathcal{Q}, \mathcal{Q}\}$ .
- If  $T_i \equiv T'_i$  for  $i \in \{1, \dots, n\}$  and  $U$  conforms to  $U'$  then the method type  $(p_1 : T_1, \dots, p_n : T_n)U$  conforms to  $(p'_1 : T'_1, \dots, p'_n : T'_n)U'$ .
- The polymorphic type  $[a_1 >: L_1 <: U_1, \dots, a_n >: L_n <: U_n]T$  conforms to the polymorphic type  $[a_1 >: L'_1 <: U'_1, \dots, a_n >: L'_n <: U'_n]T'$  if, assuming  $L'_1 <: a_1 <: U'_1, \dots, L'_n <: a_n <: U'_n$  one has  $T <: T'$  and  $L_i <: L'_i$  and  $U'_i <: U_i$  for  $i \in \{1, \dots, n\}$ .
- Type constructors  $T$  and  $T'$  follow a similar discipline. We characterize  $T$  and  $T'$  by their type parameter clauses  $[a_1, \dots, a_n]$  and  $[a'_1, \dots, a'_n]$ , where an  $a_i$  or  $a'_i$  may include a variance annotation, a higher-order type parameter clause, and bounds. Then,  $T$  conforms to  $T'$  if any list  $[t_1, \dots, t_n]$  – with declared variances, bounds and higher-order type parameter clauses – of valid type arguments for  $T'$  is also a valid list of type arguments for  $T$  and  $T[t_1, \dots, t_n] <: T'[t_1, \dots, t_n]$ . Note that this entails that:
  - The bounds on  $a_i$  must be weaker than the corresponding bounds declared for  $a'_i$ .
  - The variance of  $a_i$  must match the variance of  $a'_i$ , where covariance matches covariance, contravariance matches contravariance and any variance matches invariance.
  - Recursively, these restrictions apply to the corresponding higher-order type parameter clauses of  $a_i$  and  $a'_i$ .

A declaration or definition in some compound type of class type  $C$  *subsumes* another declaration of the same name in some compound type or class type  $C'$ , if one of the following holds.

- A value declaration or definition that defines a name  $x$  with type  $T$  subsumes a value or method declaration that defines  $x$  with type  $T'$ , provided  $T <: T'$ .
- A method declaration or definition that defines a name  $x$  with type  $T$  subsumes a method declaration that defines  $x$  with type  $T'$ , provided

$T <: T'$ .

- A type alias `type  $t[T_1, \dots, T_n] = T$`  subsumes a type alias `type  $t'[T_1, \dots, T_n] = T'$`  if  $T \equiv T'$ .
- A type declaration `type  $t[T_1, \dots, T_n] >: L <: U$`  subsumes a type declaration `type  $t'[T_1, \dots, T_n] >: L' <: U'$`  if  $L' <: L$  and  $U <: U'$ .
- A type or class definition that binds a type name  $t$  subsumes an abstract type declaration `type  $t[T_1, \dots, T_n] >: L <: U$`  if  $L <: t <: U$ .

**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 or greatest lower bound of a set of types does not always exist. For instance, consider the class definitions:

```
class A[+T] {}
class B extends A[B]
class C extends A[C]
```

Then the types  $A[Any]$ ,  $A[A[Any]]$ ,  $A[A[A[Any]]]$ ,  $\dots$  form a descending sequence of upper bounds for B and C. The least upper bound would be the infinite limit of that sequence, which does not exist as a Scala type. Since cases like this are in general impossible to detect, a Scala compiler is free to reject a term which has a type specified as a least upper or greatest lower bound, and that bound would be more complex than some compiler-set limit <sup>5</sup>.

The least upper bound or greatest lower bound might also not be unique. For instance `A with B` and `B with A` are both greatest lower bounds of A and B. If there are several least upper bounds or greatest lower bounds, the Scala compiler is free to pick any one of them.

## 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.

```
Byte  <:_w Short
Short <:_w Int
Char  <:_w Int
Int   <:_w Long
Long  <:_w Float
Float <:_w Double
```

---

<sup>5</sup>The current Scala compiler limits the nesting level of parameterization in such bounds to be at most two deeper than the maximum nesting level of the operand types



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

## 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: - view application: there's an implicit view from  $T$  to  $U$ ; - dropping by-name modifiers: if  $U$  is of the shape  $\$ \Rightarrow U'\$$  (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  $U'$ ,  $\$T <:_w U'\$$ .

## 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`.

## Volatile Types

Type volatility approximates the possibility that a type parameter or abstract type instance of a type does not have any non-null values. A value member of a volatile type cannot appear in a path.

A type is *volatile* if it falls into one of four categories:

A compound type  $\$T_1\$ \text{ with } \dots \text{ with } \$T_n\$ \{ \$R\backslash, \$\}$  is volatile if one of the following two conditions hold.

1. One of  $T_2, \dots, T_n$  is a type parameter or abstract type, or
2.  $T_1$  is an abstract type and either the refinement  $R$  or a type  $T_j$  for  $j > 1$  contributes an abstract member to the compound type, or
3. one of  $T_1, \dots, T_n$  is a singleton type.

Here, a type  $S$  *contributes an abstract member* to a type  $T$  if  $S$  contains an abstract member that is also a member of  $T$ . A refinement  $R$  contributes an abstract member to a type  $T$  if  $R$  contains an abstract declaration which is also a member of  $T$ .

A type designator is volatile if it is an alias of a volatile type, or if it designates a type parameter or abstract type that has a volatile type as its upper bound.

A singleton type  $\$p\$.type$  is volatile, if the underlying type of path  $p$  is volatile.

An existential type  $\$T\$ \text{ forSome } \{\$R\,Q\,\$, \$\}$  is volatile if  $T$  is volatile.

## 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 an alias type is the erasure of its right-hand side.
- The erasure of an abstract type is the erasure of its upper bound.
- The erasure of the parameterized type `scala.Array$[T_1]$` is `scala.Array$[|T_1|]`.
- The erasure of every other parameterized type  $T[T_1, \dots, T_n]$  is  $|T|$ .
- The erasure of a singleton type  $\$p\$.type$  is the erasure of the type of  $p$ .
- The erasure of a type projection  $\$T\#\$x\$$  is  $|\$T\$| \# \$x\$$ .
- The erasure of a compound type  $\$T_1\$ \text{ with } \$\ldots\$ \text{ with } \$T_n\$ \{\$R\,Q\,\$, \$\}$  is the erasure of the intersection dominator of  $T_1, \dots, T_n$ .
- The erasure of an existential type  $\$T\$ \text{ forSome } \{\$R\,Q\,\$, \$\}$  is  $|T|$ .

The *intersection dominator* of a list of types  $T_1, \dots, T_n$  is computed as follows. Let  $T_{i_1}, \dots, T_{i_m}$  be the subsequence of types  $T_i$  which are not supertypes of some other type  $T_j$ . If this subsequence contains a type designator  $T_c$  that refers to a class which is not a trait, the intersection dominator is  $T_c$ . Otherwise, the intersection dominator is the first element of the subsequence,  $T_{i_1}$ .