Scala 3 Reference

Internal Draft (rev 0a8bb4b199)

Martin Odersky and al.

30 December 2020



Latest HTML version available at https://dotty.epfl.ch/docs/Reference/

Contents

1	Ove	rview	5				
	1.1	Goals	5				
	1.2	Essential Foundations	5				
	1.3	Simplifications	5				
	1.4	Restrictions	6				
	1.5	Dropped Constructs	6				
	1.6	Changes	7				
	1.7	New Constructs	7				
	1.8	Metaprogramming	8				
	1.9	See Also	8				
	_	Soft Keywords	8				
2	New Types 10						
_	2.1	v -	10				
	2.2	• -	11				
	2.3	v -	11				
	$\frac{2.5}{2.4}$	V 1	12				
	2.4 2.5	V I	12 16				
	$\frac{2.5}{2.6}$	- · · · · · · · · · · · · · · · · · · ·	16				
•	Б.						
3	Enu		19				
	3.1		19				
	3.2	O T T T	21				
	3.3	Translation of Enums and ADTs	24				
4	Contextual Abstractions 29						
	4.1	Overview	29				
	4.2	Given Instances	32				
	4.3	Using Clauses	35				
	4.4	Context Bounds	38				
	4.5	Importing Givens	38				
	4.6	Extension Methods	40				
	4.7	Implementing Type classes	46				
	4.8	Type Class Derivation	50				
	4.9	Multiversal Equality	58				
	4.10	- *	63				
	4.11		66				
		1	67				
		·	68				
		1	72				
5	Mat	aprogramming 7	76				
J	5.1		76				
	5.2		76				
			70 87				

	5.4 5.5 5.6 5.7 5.8	Multi-Stage Programming10TASTy Reflect10TASTy Inspection10Erased Terms10The Meta-theory of Symmetric Metaprogramming12	03 05 06 10
6		er New Features 11	
	6.1	Trait Parameters	
	6.2	Transparent Traits	
	6.3	Universal Apply Methods	
	6.4	Export clauses	
	6.5	Opaque Type Aliases	
	6.6	Open Classes	
	6.7	Parameter Untupling	
	6.8	Kind Polymorphism	
	6.9	The Matchable Trait	
	6.10	OthreadUnsafe Annotation	
	6.11	CtargetName Annotations	
		New Control Syntax	
		Optional Braces	
		Safe Initialization	
		Named Type Arguments	
		TypeTest	
	0.1.		, ,
7	Oth	er Changed Features 15	
	7.1	Numeric Literals	
	7.2	Programmatic Structural Types	
	7.3	Rules for Operators	
	7.4	Wildcard Arguments in Types	
	7.5	Changes in Type Checking	
	7.6	Changes in Type Inference	
	7.7	Changes in Implicit Resolution	
	7.8	Implicit Conversions	
	7.9	Changes in Overload Resolution	
	7.10	Match Expressions	
	7.11	Vararg Patterns	
		Pattern Bindings	
		Option-less pattern matching	
		Automatic Eta Expansion	
		Changes in Compiler Plugins	
		Lazy Vals Initialization	
		Main Methods	
	7.18	Escapes in interpolations	ĮΙ
8	Dro	pped Features 19)2
	8.1	Dropped: DelayedInit	

	8.2	Dropped: Scala 2 Macros	192
	8.3	Dropped: Existential types	192
	8.4	Dropped: General Type Projection	193
	8.5	Dropped: Do-While	193
	8.6	Dropped: Procedure Syntax	194
	8.7	Dropped: Package Objects	194
	8.8	Dropped: Early Initializers	
	8.9	Dropped: Class shadowing	
	8.10	Dropped: Limit 22	196
	8.11	Dropped: XML literals	196
		Dropped: Symbol literals	
	8.13	Dropped: Auto-Application	197
	8.14	Dropped: Weak conformance	198
		Deprecated: Nonlocal Returns	
	8.16	Dropped: private[this] and protected[this]	199
^	a 1		200
9	Scal	a 3 Syntax Summary	200
10	App	endix	209
		Context Functions - More Details	209
		Opaque Type Aliases: More Details	
		Named Type Arguments - More Details	
		Parameter Untupling - More Details	
		Dropped: Class Shadowing - More Details	
		Dropped: Weak Conformance - More Details	
	10.7	Automatic Eta Expansion - More Details	215
	10.8	Implicit Conversions - More Details	217
	10.9	Differences with Scala 2 implicit conversions	218
	10.10	Programmatic Structural Types - More Details	219
	10.11	Dependent Function Types - More Details	221
	10.12	Intersection Types - More Details	222
	10.13	BType Lambdas - More Details	224
	10.14		000
	40 4	Union Types - More Details	226
	10.15	v -	
		Union Types - More Details	229

1 Overview

The forthcoming Scala 3 implements many language changes and improvements over Scala 2. In this reference, we discuss design decisions and present important differences compared to Scala 2.

1.1 Goals

The language redesign was guided by three main goals:

- Strengthen Scala's foundations. Make the full programming language compatible with the foundational work on the DOT calculus and apply the lessons learned from that work.
- Make Scala easier and safer to use. Tame powerful constructs such as implicits to provide a gentler learning curve. Remove warts and puzzlers.
- Further improve the consistency and expressiveness of Scala's language constructs.

Corresponding to these goals, the language changes fall into seven categories: (1) Core constructs to strengthen foundations, (2) simplifications and (3) restrictions, to make the language easier and safer to use, (4) dropped constructs to make the language smaller and more regular, (5) changed constructs to remove warts, and increase consistency and usability, (6) new constructs to fill gaps and increase expressiveness, (7) a new, principled approach to metaprogramming that replaces today's experimental macros.

1.2 Essential Foundations

These new constructs directly model core features of DOT, higher-kinded types, and the SI calculus for implicit resolution.

- Intersection types, replacing compound types,
- Union types,
- Type lambdas, replacing encodings using structural types and type projection.
- Context functions, offering abstraction over given parameters.

1.3 Simplifications

These constructs replace existing constructs with the aim of making the language safer and simpler to use, and to promote uniformity in code style.

- Trait parameters replace early initializers with a more generally useful construct.
- Given instances replace implicit objects and defs, focussing on intent over mechanism.
- Using clauses replace implicit parameters, avoiding their ambiguities.
- Extension methods replace implicit classes with a clearer and simpler mechanism.

- Opaque type aliases replace most uses of value classes while guaranteeing absence of boxing.
- Top-level definitions replace package objects, dropping syntactic boilerplate.
- Export clauses provide a simple and general way to express aggregation, which can replace the previous facade pattern of package objects inheriting from classes.
- Vararg patterns now use the form : _* instead of @ _*, mirroring vararg expressions,
- Creator applications allow using simple function call syntax instead of new expressions. new expressions stay around as a fallback for the cases where creator applications cannot be used.

With the exception of early initializers and old-style vararg patterns, all superseded constructs continue to be available in Scala 3.0. The plan is to deprecate and phase them out later.

Value classes (superseded by opaque type aliases) are a special case. There are currently no deprecation plans for value classes, since we might want to bring them back in a more general form if they are supported natively by the JVM as is planned by project Valhalla.

1.4 Restrictions

These constructs are restricted to make the language safer.

- Implicit Conversions: there is only one way to define implicit conversions instead of many, and potentially surprising implicit conversions require a language import.
- Given Imports: implicits now require a special form of import, to make the import clearly visible.
- Type Projection: only classes can be used as prefix C of a type projection C#A. Type projection on abstract types is no longer supported since it is unsound.
- Multiversal Equality implements an "opt-in" scheme to rule out nonsensical comparisons with == and !=.
- infix make method application syntax uniform across code bases.

Unrestricted implicit conversions continue to be available in Scala 3.0, but will be deprecated and removed later. Unrestricted versions of the other constructs in the list above are available only under -source 3.0-migration.

1.5 Dropped Constructs

These constructs are proposed to be dropped without a new construct replacing them. The motivation for dropping these constructs is to simplify the language and its implementation.

- DelayedInit,
- Existential types,
- Procedure syntax,

- Class shadowing,
- XML literals,
- Symbol literals,
- Auto application,
- Weak conformance,
- Compound types,
- Auto tupling (implemented, but not merged).

The date when these constructs are dropped varies. The current status is:

- Not implemented at all:
 - DelayedInit, existential types, weak conformance.
- Supported under -source 3.0-migration:
 - procedure syntax, class shadowing, symbol literals, auto application, auto tupling in a restricted form.
- Supported in 3.0, to be deprecated and phased out later:
 - XML literals, compound types.

1.6 Changes

These constructs have undergone changes to make them more regular and useful.

- Structural Types: They now allow pluggable implementations, which greatly increases their usefulness. Some usage patterns are restricted compared to the status quo.
- Name-based pattern matching: The existing undocumented Scala 2 implementation has been codified in a slightly simplified form.
- Eta expansion is now performed universally also in the absence of an expected type. The postfix _ operator is thus made redundant. It will be deprecated and dropped after Scala 3.0.
- Implicit Resolution: The implicit resolution rules have been cleaned up to make them more useful and less surprising. Implicit scope is restricted to no longer include package prefixes.

Most aspects of old-style implicit resolution are still available under -source 3.0-migration. The other changes in this list are applied unconditionally.

1.7 New Constructs

These are additions to the language that make it more powerful or pleasant to use.

- Enums provide concise syntax for enumerations and algebraic data types.
- Parameter Untupling avoids having to use case for tupled parameter destructuring.
- Dependent Function Types generalize dependent methods to dependent function values and types.
- Polymorphic Function Types generalize polymorphic methods to polymorphic function values and types. *Current status*: There is a proposal and a merged

prototype implementation, but the implementation has not been finalized (it is notably missing type inference support).

- Kind Polymorphism allows the definition of operators working equally on types and type constructors.
- @targetName Annotations make it easier to interoperate with code written in other languages and give more flexibility for avoiding name clashes.

1.8 Metaprogramming

The following constructs together aim to put metaprogramming in Scala on a new basis. So far, metaprogramming was achieved by a combination of macros and libraries such as Shapeless that were in turn based on some key macros. Current Scala 2 macro mechanisms are a thin veneer on top the current Scala 2 compiler, which makes them fragile and in many cases impossible to port to Scala 3.

It's worth noting that macros were never included in the Scala 2 language specification and were so far made available only under an -experimental flag. This has not prevented their widespread usage.

To enable porting most uses of macros, we are experimenting with the advanced language constructs listed below. These designs are more provisional than the rest of the proposed language constructs for Scala 3.0. There might still be some changes until the final release. Stabilizing the feature set needed for metaprogramming is our first priority.

- Match Types allow computation on types.
- Inline provides by itself a straightforward implementation of some simple macros and is at the same time an essential building block for the implementation of complex macros.
- Quotes and Splices provide a principled way to express macros and staging with a unified set of abstractions.
- Type class derivation provides an in-language implementation of the **Gen** macro in Shapeless and other foundational libraries. The new implementation is more robust, efficient and easier to use than the macro.
- Implicit by-name parameters provide a more robust in-language implementation of the Lazy macro in Shapeless.

1.9 See Also

A classification of proposed language features is an expanded version of this page that adds the status (i.e. relative importance to be a part of Scala 3, and relative urgency when to decide this) and expected migration cost of each language construct.

1.10 Soft Keywords

A soft modifier is one of the identifiers opaque, inline, open, transparent, and infix.

A soft keyword is a soft modifier, or one of derives, end, extension, using, | , +, -, * |

A soft modifier is treated as potential modifier of a definition if it is followed by a hard modifier or a keyword combination starting a definition (def, val, var, type, given, class, trait, object, enum, case class, case object). Between the two words there may be a sequence of newline tokens and soft modifiers.

Otherwise, soft keywords are treated specially in the following situations:

- inline, if it is followed by if, match, or a parameter definition.
- derives, if it appears after an extension clause or after the name and possibly parameters of a class, trait, object, or enum definition.
- end, if it appears at the start of a line following a statement (i.e. definition or toplevel expression)
- extension, if it appears at the start of a statement and is followed by (or [.
- using, if it appears at the start of a parameter or argument list.
- |, if it separates two patterns in an alternative.
- +, -, if they appear in front of a type parameter.
- *, if it follows the type of a parameter or if it appears in a vararg type ascription
 x: _*.

Everywhere else a soft keyword is treated as a normal identifier.

2 New Types 2

2.1 Intersection types

Used on types, the & operator creates an intersection type.

2.1.1 Type Checking

The type S & T represents values that are of the type S and T at the same time.

```
trait Resettable:
    def reset(): Unit

trait Growable[T]:
    def add(t: T): Unit

def f(x: Resettable & Growable[String]) =
    x.reset()
    x.add("first")
```

The parameter x is required to be both a Resettable and a Growable [String].

The members of an intersection type A & B are all the members of A and all the members of B. For instance Resettable & Growable[String] has member methods reset and add.

& is commutative: A & B is the same type as B & A.

If a member appears in both A and B, its type in A & B is the intersection of its type in A and its type in B. For instance, assume the definitions:

```
trait A:
    def children: List[A]

trait B:
    def children: List[B]

val x: A & B = new C
val ys: List[A & B] = x.children
```

The type of children in A & B is the intersection of children's type in A and its type in B, which is List[A] & List[B]. This can be further simplified to List[A & B] because List is covariant.

One might wonder how the compiler could come up with a definition for children of type List[A & B] since what is given are children definitions of type List[A] and List[B]. The answer is the compiler does not need to. A & B is just a type that represents a set of requirements for values of the type. At the point where a value is *constructed*, one must make sure that all inherited members are correctly defined. So if one defines a class C that inherits A and B, one needs to give at that point a definition of a children method with the required type.

```
class C extends A, B:
  def children: List[A & B] = ???
```

More details

2.2 Union types

A union type A | B has as values all values of type A and also all values of type B.

```
case class UserName(name: String)
case class Password(hash: Hash)

def help(id: UserName | Password) =
  val user = id match
     case UserName(name) => lookupName(name)
     case Password(hash) => lookupPassword(hash)
```

Union types are duals of intersection types. | is *commutative*: A | B is the same type as B | A.

The compiler will assign a union type to an expression only if such a type is explicitly given. This can be seen in the following REPL transcript:

```
scala> val password = Password(123)
val password: Password = Password(123)

scala> val name = UserName("Eve")
val name: UserName = UserName(Eve)

scala> if true then name else password
val res2: Object & Product = UserName(Eve)

scala> val either: Password | UserName = if true then name else password
val either: Password | UserName = UserName(Eve)
```

The type of res2 is Object & Product, which is a supertype of UserName and Password, but not the least supertype Password | UserName. If we want the least supertype, we have to give it explicitly, as is done for the type of either.

More details

2.3 Type lambdas

A type lambda lets one express a higher-kinded type directly, without a type definition.

```
[X, Y] \implies Map[Y, X]
```

For instance, the type above defines a binary type constructor, which maps arguments X and Y to Map[Y, X]. Type parameters of type lambdas can have bounds,

but they cannot carry + or - variance annotations.

More details

2.4 Match Types

A match type reduces to one of its right-hand sides, depending on the type of its scrutinee. For example:

```
type Elem[X] = X match
  case String => Char
  case Array[t] => t
  case Iterable[t] => t
```

This defines a type that reduces as follows:

```
Elem[String] =:= Char
Elem[Array[Int]] =:= Int
Elem[List[Float]] =:= Float
Elem[Nil.type] =:= Nothing
```

Here =:= is understood to mean that left and right hand sides are mutually subtypes of each other.

In general, a match type is of the form

```
S match \{ P1 \Rightarrow T1 \dots Pn \Rightarrow Tn \}
```

where S, T1, ..., Tn are types and P1, ..., Pn are type patterns. Type variables in patterns start with a lower case letter, as usual.

Match types can form part of recursive type definitions. Example:

```
type LeafElem[X] = X match
  case String => Char
  case Array[t] => LeafElem[t]
  case Iterable[t] => LeafElem[t]
  case AnyVal => X
```

Recursive match type definitions can also be given an upper bound, like this:

```
type Concat[Xs <: Tuple, +Ys <: Tuple] <: Tuple = Xs match
  case Unit => Ys
  case x *: xs => x *: Concat[xs, Ys]
```

In this definition, every instance of Concat[A, B], whether reducible or not, is known to be a subtype of Tuple. This is necessary to make the recursive invocation x *: Concat[xs, Ys] type check, since *: demands a Tuple as its right operand.

2.4.1 Dependent Typing

Match types can be used to define dependently typed methods. For instance, here is the value level counterpart to the LeafElem type defined above (note the use of

the match type as the return type):

```
def leafElem[X](x: X): LeafElem[X] = x match
  case x: String => x.charAt(0)
  case x: Array[t] => leafElem(x(9))
  case x: Iterable[t] => leafElem(x.next())
  case x: AnyVal => x
```

This special mode of typing for match expressions is only used when the following conditions are met:

- 1. The match expression patterns do not have guards
- 2. The match expression scrutinee's type is a subtype of the match type scrutinee's type
- 3. The match expression and the match type have the same number of cases
- 4. The match expression patterns are all Typed Patterns, and these types are =:= to their corresponding type patterns in the match type

2.4.2 Representation of Match Types

The internal representation of a match type

```
S match { P1 => T1 ... Pn => Tn }
is Match(S, C1, ..., Cn) <: B where each case Ci is of the form
[Xs] =>> P => T
```

Here, [Xs] is a type parameter clause of the variables bound in pattern Pi. If there are no bound type variables in a case, the type parameter clause is omitted and only the function type P => T is kept. So each case is either a unary function type or a type lambda over a unary function type.

B is the declared upper bound of the match type, or Any if no such bound is given. We will leave it out in places where it does not matter for the discussion. The scrutinee, bound, and pattern types must all be first-order types.

2.4.3 Match Type Reduction

Match type reduction follows the semantics of match expressions, that is, a match type of the form S match { P1 => T1 ... Pn => Tn } reduces to Ti if and only if s: S match { $_{\cdot}$: P1 => T1 ... $_{\cdot}$: Pn => Tn } evaluates to a value of type Ti for all s: S.

The compiler implements the following reduction algorithm:

- If the scrutinee type S is an empty set of values (such as Nothing or String & Int), do not reduce.
- Sequentially consider each pattern Pi
 - − If S <: Pi reduce to Ti.
 - Otherwise, try constructing a proof that S and Pi are disjoint, or, in other words, that no value s of type S is also of type Pi.

 If such proof is found, proceed to the next case (Pi+1), otherwise, do not reduce.

Disjointness proofs rely on the following properties of Scala types:

- 1. Single inheritance of classes
- 2. Final classes cannot be extended
- 3. Constant types with distinct values are nonintersecting

Type parameters in patterns are minimally instantiated when computing S <: Pi. An instantiation Is is *minimal* for Xs if all type variables in Xs that appear covariantly and nonvariantly in Is are as small as possible and all type variables in Xs that appear contravariantly in Is are as large as possible. Here, "small" and "large" are understood with respect to <:.

For simplicity, we have omitted constraint handling so far. The full formulation of subtyping tests describes them as a function from a constraint and a pair of types to either *success* and a new constraint or *failure*. In the context of reduction, the subtyping test S <: [Xs := Is] P is understood to leave the bounds of all variables in the input constraint unchanged, i.e. existing variables in the constraint cannot be instantiated by matching the scrutinee against the patterns.

2.4.4 Subtyping Rules for Match Types

The following rules apply to match types. For simplicity, we omit environments and constraints.

1. The first rule is a structural comparison between two match types:

```
S match { P1 => T1 ... Pm => Tm } <: T match { Q1 => U1 ... Qn => Un } if S =:= T, \quad m >= n, \quad Pi =:= Qi \ and \ Ti <: \ Ui \ for \ i \ in \ 1..n
```

I.e. scrutinees and patterns must be equal and the corresponding bodies must be subtypes. No case re-ordering is allowed, but the subtype can have more cases than the supertype.

2. The second rule states that a match type and its redux are mutual subtypes.

```
S match { P1 => T1 ... Pn => Tn } <: U
U <: S match { P1 => T1 ... Pn => Tn }
if
S match { P1 => T1 ... Pn => Tn } reduces to U
```

3. The third rule states that a match type conforms to its upper bound:

```
(S match { P1 => T1 ... Pn => Tn } <: B) <: B
```

2.4.5 Termination

Match type definitions can be recursive, which means that it's possible to run into an infinite loop while reducing match types.

Since reduction is linked to subtyping, we already have a cycle detection mechanism in place. As a result, the following will already give a reasonable error message:

Internally, the Scala compiler detects these cycles by turning selected stack overflows into type errors. If there is a stack overflow during subtyping, the exception will be caught and turned into a compile-time error that indicates a trace of the subtype tests that caused the overflow without showing a full stack trace.

2.4.6 Variance Laws for Match Types

NOTE: This section does not reflect the current implementation.

Within a match type Match(S, Cs) <: B, all occurrences of type variables count as covariant. By the nature of the cases Ci this means that occurrences in pattern position are contravariant (since patterns are represented as function type arguments).

2.4.7 Related Work

Match types have similarities with closed type families in Haskell. Some differences are:

- Subtyping instead of type equalities.
- Match type reduction does not tighten the underlying constraint, whereas type family reduction does unify. This difference in approach mirrors the difference between local type inference in Scala and global type inference in Haskell.

Match types are also similar to Typescript's conditional types. The main differences here are:

• Conditional types only reduce if both the scrutinee and pattern are ground, whereas match types also work for type parameters and abstract types.

- Match types support direct recursion.
- Conditional types distribute through union types.

2.5 Dependent Function Types

A dependent function type is a function type whose result depends on the function's parameters. For example:

Scala already has dependent methods, i.e. methods where the result type refers to some of the parameters of the method. Method extractKey is an example. Its result type, e.Key refers to its parameter e (we also say, e.Key depends on e). But so far it was not possible to turn such methods into function values, so that they can be passed as parameters to other functions, or returned as results. Dependent methods could not be turned into functions simply because there was no type that could describe them.

In Scala 3 this is now possible. The type of the extractor value above is

```
(e: Entry) => e.Key
```

This type describes function values that take any argument e of type Entry and return a result of type e.Key.

Recall that a normal function type $A \Rightarrow B$ is represented as an instance of the Function1 trait (i.e. Function1[A, B]) and analogously for functions with more parameters. Dependent functions are also represented as instances of these traits, but they get an additional refinement. In fact, the dependent function type above is just syntactic sugar for

```
Function1[Entry, Entry#Key]:
   def apply(e: Entry): e.Key
```

More details

2.6 Polymorphic Function Types

A polymorphic function type is a function type which accepts type parameters. For example:

```
// A polymorphic method:
def foo[A](xs: List[A]): List[A] = xs.reverse
// A polymorphic function value:
```

```
val bar: [A] => List[A] => List[A]
//

a polymorphic function type
= [A] => (xs: List[A]) => foo[A](xs)
```

Scala already has *polymorphic methods*, i.e. methods which accepts type parameters. Method **foo** above is an example, accepting a type parameter A. So far, it was not possible to turn such methods into polymorphic function values like **bar** above, which can be passed as parameters to other functions, or returned as results.

In Scala 3 this is now possible. The type of the bar value above is

```
[A] => List[A] => List[A]
```

This type describes function values which take a type A as a parameter, then take a list of type List[A], and return a list of the same type List[A].

More details

2.6.1 Example Usage

Polymorphic function type are particularly useful when callers of a method are required to provide a function which has to be polymorphic, meaning that it should accept arbitrary types as part of its inputs.

For instance, consider the situation where we have a data type to represent the expressions of a simple language (consisting only of variables and function applications) in a strongly-typed way:

```
enum Expr[A]:
   case Var(name: String)
   case Apply[A, B](fun: Expr[B => A], arg: Expr[B]) extends Expr[A]
```

We would like to provide a way for users to map a function over all immediate subexpressions of a given Expr. This requires the given function to be polymorphic, since each subexpression may have a different type. Here is how to implement this using polymorphic function types:

```
def mapSubexpressions[A](e: Expr[A])(f: [B] => Expr[B] => Expr[B]): Expr[A] =
  e match
    case Apply(fun, arg) => Apply(f(fun), f(arg))
    case Var(n) => Var(n)
```

And here is how to use this function to *wrap* each subexpression in a given expression with a call to some wrap function, defined as a variable:

```
val e0 = Apply(Var("f"), Var("a"))
val e1 = mapSubexpressions(e0)(
   [B] => (se: Expr[B]) => Apply(Var[B => B]("wrap"), se))
println(e1) // Apply(Apply(Var(wrap), Var(f)), Apply(Var(wrap), Var(a)))
```

2.6.2 Relationship With Type Lambdas

Polymorphic function types are not to be confused with *type lambdas*. While the former describes the *type* of a polymorphic *value*, the latter is an actual function value at the type level.

A good way of understanding the difference is to notice that $type\ lambdas$ are applied in types, whereas polymorphic functions are applied in terms: One would call the function bar above by passing it a type argument bar[Int] within a method body. On the other hand, given a type lambda such as type $F = [A] \implies List[A]$, one would call F within a type expression, as in type Bar = F[Int].

3 Enums 2

3.1 Enumerations

An enumeration is used to define a type consisting of a set of named values.

```
enum Color:
   case Red, Green, Blue
```

This defines a new sealed class, Color, with three values, Color.Red, Color.Green, Color.Blue. The color values are members of Colors companion object.

3.1.0.1 Parameterized enums Enums can be parameterized.

```
enum Color(val rgb: Int):
   case Red     extends Color(0xFF0000)
   case Green extends Color(0x00FF00)
   case Blue     extends Color(0x0000FF)
```

As the example shows, you can define the parameter value by using an explicit extends clause.

3.1.0.2 Methods defined for enums The values of an enum correspond to unique integers. The integer associated with an enum value is returned by its ordinal method:

```
scala> val red = Color.Red
val red: Color = Red
scala> red.ordinal
val res0: Int = 0
```

The companion object of an enum also defines three utility methods. The valueOf method obtains an enum value by its name. The values method returns all enum values defined in an enumeration in an Array. The fromOrdinal method obtains an enum value from its ordinal (Int) value.

```
scala> Color.valueOf("Blue")
val res0: Color = Blue
scala> Color.values
val res1: Array[Color] = Array(Red, Green, Blue)
scala> Color.fromOrdinal(0)
val res2: Color = Red
```

3.1.0.3 User-defined members of enums It is possible to add your own definitions to an enum. Example:

```
enum Planet(mass: Double, radius: Double):
   private final val G = 6.67300E-11
   def surfaceGravity = G * mass / (radius * radius)
   def surfaceWeight(otherMass: Double) = otherMass * surfaceGravity
```

```
case Mercury extends Planet(3.303e+23, 2.4397e6)
case Venus extends Planet(4.869e+24, 6.0518e6)
case Earth extends Planet(5.976e+24, 6.37814e6)
case Mars extends Planet(6.421e+23, 3.3972e6)
case Jupiter extends Planet(1.9e+27, 7.1492e7)
case Saturn extends Planet(5.688e+26, 6.0268e7)
case Uranus extends Planet(8.686e+25, 2.5559e7)
case Neptune extends Planet(1.024e+26, 2.4746e7)
end Planet
```

It is also possible to define an explicit companion object for an enum:

```
object Planet:
```

```
def main(args: Array[String]) =
   val earthWeight = args(0).toDouble
   val mass = earthWeight / Earth.surfaceGravity
   for p <- values do
        println(s"Your weight on $p is ${p.surfaceWeight(mass)}")
end Planet</pre>
```

3.1.0.4 Compatibility with Java Enums If you want to use the Scala-defined enums as Java enums, you can do so by extending the class java.lang.Enum, which is imported by default, as follows:

```
enum Color extends Enum[Color] { case Red, Green, Blue }
```

The type parameter comes from the Java enum definition and should be the same as the type of the enum. There is no need to provide constructor arguments (as defined in the Java API docs) to java.lang.Enum when extending it — the compiler will generate them automatically.

After defining Color like that, you can use it like you would a Java enum:

```
scala> Color.Red.compareTo(Color.Green)
val res15: Int = -1
```

For a more in-depth example of using Scala 3 enums from Java, see this test. In the test, the enums are defined in the MainScala.scala file and used from a Java source, Test.java.

3.1.0.5 Implementation Enums are represented as sealed classes that extend the scala.reflect.Enum trait. This trait defines a single public method, ordinal:

```
package scala.reflect
```

```
/** A base trait of all Scala enum definitions */
transparent trait Enum extends Any, Product, Serializable:
```

```
/** A number uniquely identifying a case of an enum */
def ordinal: Int
```

Enum values with **extends** clauses get expanded to anonymous class instances. For instance, the **Venus** value above would be defined like this:

```
val Venus: Planet = new Planet(4.869E24, 6051800.0):
   def ordinal: Int = 1
   override def productPrefix: String = "Venus"
   override def toString: String = "Venus"
```

Enum values without **extends** clauses all share a single implementation that can be instantiated using a private method that takes a tag and a name as arguments. For instance, the first definition of value Color.Red above would expand to:

```
val Red: Color = $new(0, "Red")
```

3.1.0.6 Reference For more information, see Issue #1970 and PR #4003.

3.2 Algebraic Data Types

The enum concept is general enough to also support algebraic data types (ADTs) and their generalized version (GADTs). Here is an example how an Option type can be represented as an ADT:

```
enum Option[+T]:
    case Some(x: T)
    case None
```

This example introduces an Option enum with a covariant type parameter T consisting of two cases, Some and None. Some is parameterized with a value parameter x. It is a shorthand for writing a case class that extends Option. Since None is not parameterized, it is treated as a normal enum value.

The **extends** clauses that were omitted in the example above can also be given explicitly:

Note that the parent type of the None value is inferred as Option[Nothing]. Generally, all covariant type parameters of the enum class are minimized in a compiler-generated extends clause whereas all contravariant type parameters are maximized. If Option was non-variant, you would need to give the extends clause of None explicitly.

As for normal enum values, the cases of an **enum** are all defined in the **enum**s companion object. So it's Option.Some and Option.None unless the definitions are "pulled out" with an import:

```
scala> Option.Some("hello")
val res1: t2.Option[String] = Some(hello)
scala> Option.None
val res2: t2.Option[Nothing] = None
```

Note that the type of the expressions above is always Option. Generally, the type of a enum case constructor application will be widened to the underlying enum type, unless a more specific type is expected. This is a subtle difference with respect to normal case classes. The classes making up the cases do exist, and can be unveiled, either by constructing them directly with a new, or by explicitly providing an expected type.

```
scala> new Option.Some(2)
val res3: Option.Some[Int] = Some(2)
scala> val x: Option.Some[Int] = Option.Some(3)
val res4: Option.Some[Int] = Some(3)
```

As all other enums, ADTs can define methods. For instance, here is Option again, with an isDefined method and an Option(...) constructor in its companion object.

```
enum Option[+T]:
    case Some(x: T)
    case None

def isDefined: Boolean = this match
        case None => false
        case some => true

object Option:

def apply[T >: Null](x: T): Option[T] =
    if x == null then None else Some(x)
```

Enumerations and ADTs have been presented as two different concepts. But since they share the same syntactic construct, they can be seen simply as two ends of a spectrum and it is perfectly possible to construct hybrids. For instance, the code below gives an implementation of Color either with three enum values or with a parameterized case that takes an RGB value.

```
enum Color(val rgb: Int):
    case Red        extends Color(0xFF0000)
    case Green extends Color(0x00FF00)
    case Blue        extends Color(0x0000FF)
    case Mix(mix: Int)        extends Color(mix)
```

end Option

3.2.0.1 Parameter Variance of Enums By default, parameterized cases of enums with type parameters will copy the type parameters of their parent, along with any variance notations. As usual, it is important to use type parameters carefully when they are variant, as shown below:

The following View enum has a contravariant type parameter T and a single case Refl, representing a function mapping a type T to itself:

```
enum View[-T]:
    case Refl(f: T => T)
```

The definition of Refl is incorrect, as it uses contravariant type T in the covariant result position of a function type, leading to the following error:

```
-- Error: View.scala:2:12 ------
2 | case Refl(f: T => T)
```

|contravariant type T occurs in covariant position in type T => T of value f |enum case Refl requires explicit declaration of type T to resolve this issue.

Because Refl does not declare explicit parameters, it looks to the compiler like the following:

```
enum View[-T]:
   case Refl[/*synthetic*/-T1](f: T1 => T1) extends View[T1]
```

The compiler has inferred for Refl the contravariant type parameter T1, following T in View. We can now clearly see that Refl needs to declare its own non-variant type parameter to correctly type f, and can remedy the error by the following change to Refl:

```
enum View[-T]:
- case Refl(f: T => T)
+ case Refl[R](f: R => R) extends View[R]
```

Above, type R is chosen as the parameter for Refl to highlight that it has a different meaning to type T in View, but any name will do.

After some further changes, a more complete implementation of View can be given as follows and be used as the function type $T \Rightarrow U$:

```
enum View[-T, +U] extends (T => U):
   case Refl[R](f: R => R) extends View[R, R]

final def apply(t: T): U = this match
   case refl: Refl[r] => refl.f(t)
```

- **3.2.0.2** Syntax of Enums Changes to the syntax fall in two categories: enum definitions and cases inside enums. The changes are specified below as deltas with respect to the Scala syntax given here
 - 1. Enum definitions are defined as follows:

2. Cases of enums are defined as follows:

```
EnumCase ::= `case' (id ClassConstr [`extends' ConstrApps]] | ids)
```

3.2.0.3 Reference For more information, see Issue #1970.

3.3 Translation of Enums and ADTs

The compiler expands enums and their cases to code that only uses Scala's other language features. As such, enums in Scala are convenient *syntactic sugar*, but they are not essential to understand Scala's core.

We now explain the expansion of enums in detail. First, some terminology and notational conventions:

- We use E as a name of an enum, and C as a name of a case that appears in E.
- We use <...> for syntactic constructs that in some circumstances might be empty. For instance, <value-params> represents one or more parameter lists (...) or nothing at all.
- Enum cases fall into three categories:
 - Class cases are those cases that are parameterized, either with a type parameter section [...] or with one or more (possibly empty) parameter sections (...).
 - Simple cases are cases of a non-generic enum that have neither parameters nor an extends clause or body. That is, they consist of a name only.
 - Value cases are all cases that do not have a parameter section but that do have a (possibly generated) extends clause and/or a body.

Simple cases and value cases are collectively called *singleton cases*.

The desugaring rules imply that class cases are mapped to case classes, and singleton cases are mapped to val definitions.

There are nine desugaring rules. Rule (1) desugars enum definitions. Rules (2) and (3) desugar simple cases. Rules (4) to (6) define **extends** clauses for cases that are missing them. Rules (7) to (9) define how such cases with **extends** clauses map into case classes or vals.

1. An enum definition

```
enum E ... { <defs> <cases> }
```

expands to a sealed abstract class that extends the scala.reflect.Enum trait and an associated companion object that contains the defined cases, expanded according to rules (2 - 8). The enum class starts with a compiler-generated import that imports the names <caseIds> of all cases so that they can be used without prefix in the class.

2. A simple case consisting of a comma-separated list of enum names

```
case C_1, ..., C_n
expands to
case C_1; ...; case C_n
```

Any modifiers or annotations on the original case extend to all expanded cases.

3. A simple case

```
case C
```

of an enum E that does not take type parameters expands to

```
val C = new(n, "C")
```

Here, \$new is a private method that creates an instance of E (see below).

4. If E is an enum with type parameters

where each of the variances Vi is either '+' or '-', then a simple case

case C

expands to

```
case C extends E[B1, ..., Bn]
```

where Bi is Li if Vi = '+' and Ui if Vi = '-'. This result is then further rewritten with rule (8). Simple cases of enums with non-variant type parameters are not permitted (however value cases with explicit extends clause are)

5. A class case without an extends clause

```
case C <type-params> <value-params>
```

of an enum E that does not take type parameters expands to

```
case C <type-params> <value-params> extends E
```

This result is then further rewritten with rule (9).

6. If E is an enum with type parameters Ts, a class case with neither type parameters nor an extends clause

```
case C <value-params>
expands to
```

```
case C[Ts] <value-params> extends E[Ts]
```

This result is then further rewritten with rule (9). For class cases that have type parameters themselves, an extends clause needs to be given explicitly.

7. If E is an enum with type parameters Ts, a class case without type parameters but with an extends clause

```
case C <value-params> extends <parents>
expands to
case C[Ts] <value-params> extends <parents>
```

provided at least one of the parameters Ts is mentioned in a parameter type in <value-params> or in a type argument in parents>.

8. A value case

```
case C extends <parents>
```

expands to a value definition in E's companion object:

```
val C = new <parents> { <body>; def ordinal = n }
```

where n is the ordinal number of the case in the companion object, starting from 0. The anonymous class also implements the abstract Product methods that it inherits from Enum.

It is an error if a value case refers to a type parameter of the enclosing enum in a type argument of cparents>.

9. A class case

```
case C <params> extends <parents>
```

expands analogous to a final case class in E's companion object:

```
final case class C <params> extends <parents>
```

The enum case defines an ordinal method of the form

```
def ordinal = n
```

where n is the ordinal number of the case in the companion object, starting from 0.

It is an error if a value case refers to a type parameter of the enclosing enum in a parameter type in cparams or in a type argument of cparents, unless that parameter is already a type parameter of the case, i.e. the parameter name is defined in cparams.

The compiler-generated apply and copy methods of an enum case

```
case C(ps) extends P1, ..., Pn
```

are treated specially. A call C(ts) of the apply method is ascribed the underlying type P1 & ... & Pn (dropping any transparent traits) as long as that type is still compatible with the expected type at the point of application. A call t.copy(ts) of C's copy method is treated in the same way.

- **3.3.0.1** Translation of Enums with Singleton Cases An enum E (possibly generic) that defines one or more singleton cases will define the following additional synthetic members in its companion object (where E' denotes E with any type parameters replaced by wildcards):
 - A method valueOf(name: String): E'. It returns the singleton case value whose identifier is name.
 - A method values which returns an Array[E'] of all singleton case values defined by E, in the order of their definitions.

If E contains at least one simple case, its companion object will define in addition:

 A private method \$new which defines a new simple case value with given ordinal number and name. This method can be thought as being defined as follows.

```
private def $new(_$ordinal: Int, $name: String) =
  new E with runtime.EnumValue:
    def ordinal = _$ordinal
    override def productPrefix = $name // if not overridden in `E`
    override def toString = $name // if not overridden in `E`
```

The anonymous class also implements the abstract Product methods that it inherits from Enum. The ordinal method is only generated if the enum does not extend from java.lang.Enum (as Scala enums do not extend java.lang.Enums unless explicitly specified). In case it does, there is no need to generate ordinal as java.lang.Enum defines it. Similarly there is no need to override toString as that is defined in terms of name in java.lang.Enum. Finally, productPrefix will call this.name when E extends java.lang.Enum.

3.3.0.2 Scopes for Enum Cases A case in an enum is treated similarly to a secondary constructor. It can access neither the enclosing enum using this, nor its value parameters or instance members using simple identifiers.

Even though translated enum cases are located in the enum's companion object, referencing this object or its members via this or a simple identifier is also illegal. The compiler typechecks enum cases in the scope of the enclosing companion object but flags any such illegal accesses as errors.

3.3.0.3 Translation of Java-compatible enums A Java-compatible enum is an enum that extends java.lang.Enum. The translation rules are the same as above,

with the reservations defined in this section.

It is a compile-time error for a Java-compatible enum to have class cases.

Cases such as **case** C expand to a **@static val** as opposed to a **val**. This allows them to be generated as static fields of the enum type, thus ensuring they are represented the same way as Java enums.

3.3.0.4 Other Rules

- A normal case class which is not produced from an enum case is not allowed to extend scala.reflect.Enum. This ensures that the only cases of an enum are the ones that are explicitly declared in it.
- If an enum case has an **extends** clause, the enum class must be one of the classes that's extended.

4 Contextual Abstractions

4.1 Overview

4.1.0.1 Critique of the Status Quo Scala's implicits are its most distinguished feature. They are *the* fundamental way to abstract over context. They represent a unified paradigm with a great variety of use cases, among them: implementing type classes, establishing context, dependency injection, expressing capabilities, computing new types and proving relationships between them.

Following Haskell, Scala was the second popular language to have some form of implicits. Other languages have followed suit. E.g Rust's traits or Swift's protocol extensions. Design proposals are also on the table for Kotlin as compile time dependency resolution, for C# as Shapes and Extensions or for F# as Traits. Implicits are also a common feature of theorem provers such as Coq or Agda.

Even though these designs use widely different terminology, they are all variants of the core idea of term inference. Given a type, the compiler synthesizes a "canonical" term that has that type. Scala embodies the idea in a purer form than most other languages: An implicit parameter directly leads to an inferred argument term that could also be written down explicitly. By contrast, type class based designs are less direct since they hide term inference behind some form of type classification and do not offer the option of writing the inferred quantities (typically, dictionaries) explicitly.

Given that term inference is where the industry is heading, and given that Scala has it in a very pure form, how come implicits are not more popular? In fact, it's fair to say that implicits are at the same time Scala's most distinguished and most controversial feature. I believe this is due to a number of aspects that together make implicits harder to learn than necessary and also make it harder to prevent abuses.

Particular criticisms are:

1. Being very powerful, implicits are easily over-used and mis-used. This observation holds in almost all cases when we talk about *implicit conversions*, which, even though conceptually different, share the same syntax with other implicit definitions. For instance, regarding the two definitions

```
implicit def i1(implicit x: T): C[T] = ... implicit def i2(x: T): C[T] = ...
```

the first of these is a conditional implicit value, the second an implicit conversion. Conditional implicit values are a cornerstone for expressing type classes, whereas most applications of implicit conversions have turned out to be of dubious value. The problem is that many newcomers to the language start with defining implicit conversions since they are easy to understand and seem powerful and convenient. Scala 3 will put under a language flag both definitions and applications of "undisciplined" implicit conversions between types defined elsewhere. This is a useful step to push back against overuse of implicit conversions. But the problem remains that syntactically, conversions

and values just look too similar for comfort.

- 2. Another widespread abuse is over-reliance on implicit imports. This often leads to inscrutable type errors that go away with the right import incantation, leaving a feeling of frustration. Conversely, it is hard to see what implicits a program uses since implicits can hide anywhere in a long list of imports.
- 3. The syntax of implicit definitions is too minimal. It consists of a single modifier, implicit, that can be attached to a large number of language constructs. A problem with this for newcomers is that it conveys mechanism instead of intent. For instance, a type class instance is an implicit object or val if unconditional and an implicit def with implicit parameters referring to some class if conditional. This describes precisely what the implicit definitions translate to just drop the implicit modifier, and that's it! But the cues that define intent are rather indirect and can be easily misread, as demonstrated by the definitions of i1 and i2 above.
- 4. The syntax of implicit parameters also has shortcomings. While implicit parameters are designated specifically, arguments are not. Passing an argument to an implicit parameter looks like a regular application f(arg). This is problematic because it means there can be confusion regarding what parameter gets instantiated in a call. For instance, in

```
def currentMap(implicit ctx: Context): Map[String, Int]
```

one cannot write currentMap("abc") since the string "abc" is taken as explicit argument to the implicit ctx parameter. One has to write currentMap.apply("abc") instead, which is awkward and irregular. For the same reason, a method definition can only have one implicit parameter section and it must always come last. This restriction not only reduces orthogonality, but also prevents some useful program constructs, such as a method with a regular parameter whose type depends on an implicit value. Finally, it's also a bit annoying that implicit parameters must have a name, even though in many cases that name is never referenced.

5. Implicits pose challenges for tooling. The set of available implicits depends on context, so command completion has to take context into account. This is feasible in an IDE but tools like Scaladoc that are based on static web pages can only provide an approximation. Another problem is that failed implicit searches often give very unspecific error messages, in particular if some deeply recursive implicit search has failed. Note that the Scala 3 compiler has already made a lot of progress in the error diagnostics area. If a recursive search fails some levels down, it shows what was constructed and what is missing. Also, it suggests imports that can bring missing implicits in scope.

None of the shortcomings is fatal, after all implicits are very widely used, and many libraries and applications rely on them. But together, they make code using implicits a lot more cumbersome and less clear than it could be.

Historically, many of these shortcomings come from the way implicits were gradu-

ally "discovered" in Scala. Scala originally had only implicit conversions with the intended use case of "extending" a class or trait after it was defined, i.e. what is expressed by implicit classes in later versions of Scala. Implicit parameters and instance definitions came later in 2006 and we picked similar syntax since it seemed convenient. For the same reason, no effort was made to distinguish implicit imports or arguments from normal ones.

Existing Scala programmers by and large have gotten used to the status quo and see little need for change. But for newcomers this status quo presents a big hurdle. I believe if we want to overcome that hurdle, we should take a step back and allow ourselves to consider a radically new design.

4.1.0.2 The New Design The following pages introduce a redesign of contextual abstractions in Scala. They introduce four fundamental changes:

- 1. Given Instances are a new way to define basic terms that can be synthesized. They replace implicit definitions. The core principle of the proposal is that, rather than mixing the implicit modifier with a large number of features, we have a single way to define terms that can be synthesized for types.
- 2. Using Clauses are a new syntax for implicit parameters and their arguments. It unambiguously aligns parameters and arguments, solving a number of language warts. It also allows us to have several using clauses in a definition.
- 3. "Given" Imports are a new class of import selectors that specifically import givens and nothing else.
- 4. Implicit Conversions are now expressed as given instances of a standard Conversion class. All other forms of implicit conversions will be phased out.

This section also contains pages describing other language features that are related to context abstraction. These are:

- Context Bounds, which carry over unchanged.
- Extension Methods replace implicit classes in a way that integrates better with type classes.
- Implementing Type Classes demonstrates how some common type classes can be implemented using the new constructs.
- Type Class Derivation introduces constructs to automatically derive type class instances for ADTs.
- Multiversal Equality introduces a special type class to support type safe equality.
- Context Functions provide a way to abstract over context parameters.
- By-Name Context Parameters are an essential tool to define recursive synthesized values without looping.
- Relationship with Scala 2 Implicits discusses the relationship between old-style implicits and new-style givens and how to migrate from one to the other.

Overall, the new design achieves a better separation of term inference from the rest of the language: There is a single way to define givens instead of a multitude of forms all taking an implicit modifier. There is a single way to introduce implicit parameters and arguments instead of conflating implicit with normal arguments. There is a separate way to import givens that does not allow them to hide in a sea of normal imports. And there is a single way to define an implicit conversion which is clearly marked as such and does not require special syntax.

This design thus avoids feature interactions and makes the language more consistent and orthogonal. It will make implicits easier to learn and harder to abuse. It will greatly improve the clarity of the 95% of Scala programs that use implicits. It has thus the potential to fulfil the promise of term inference in a principled way that is also accessible and friendly.

Could we achieve the same goals by tweaking existing implicits? After having tried for a long time, I believe now that this is impossible.

- First, some of the problems are clearly syntactic and require different syntax to solve them.
- Second, there is the problem how to migrate. We cannot change the rules in mid-flight. At some stage of language evolution we need to accommodate both the new and the old rules. With a syntax change, this is easy: Introduce the new syntax with new rules, support the old syntax for a while to facilitate cross compilation, deprecate and phase out the old syntax at some later time. Keeping the same syntax does not offer this path, and in fact does not seem to offer any viable path for evolution
- Third, even if we would somehow succeed with migration, we still have the problem how to teach this. We cannot make existing tutorials go away. Almost all existing tutorials start with implicit conversions, which will go away; they use normal imports, which will go away, and they explain calls to methods with implicit parameters by expanding them to plain applications, which will also go away. This means that we'd have to add modifications and qualifications to all existing literature and courseware, likely causing more confusion with beginners instead of less. By contrast, with a new syntax there is a clear criterion: Any book or courseware that mentions implicit is outdated and should be updated.

4.2 Given Instances

Given instances (or, simply, "givens") define "canonical" values of certain types that serve for synthesizing arguments to context parameters. Example:

```
trait Ord[T]:
    def compare(x: T, y: T): Int
    extension (x: T) def < (y: T) = compare(x, y) < 0
    extension (x: T) def > (y: T) = compare(x, y) > 0

given intOrd: Ord[Int] with
    def compare(x: Int, y: Int) =
        if x < y then -1 else if x > y then +1 else 0
```

```
given listOrd[T](using ord: Ord[T]): Ord[List[T]] with

def compare(xs: List[T], ys: List[T]): Int = (xs, ys) match
    case (Nil, Nil) => 0
    case (Nil, _) => -1
    case (_, Nil) => +1
    case (x :: xs1, y :: ys1) =>
        val fst = ord.compare(x, y)
        if fst != 0 then fst else compare(xs1, ys1)
```

This code defines a trait Ord with two given instances. intOrd defines a given for the type Ord[Int] whereas listOrd[T] defines givens for Ord[List[T]] for all types T that come with a given instance for Ord[T] themselves. The using clause in listOrd defines a condition: There must be a given of type Ord[T] for a given of type Ord[List[T]] to exist. Such conditions are expanded by the compiler to context parameters.

4.2.1 Anonymous Givens

The name of a given can be left out. So the definitions of the last section can also be expressed like this:

```
given Ord[Int] with
...
given [T](using Ord[T]): Ord[List[T]] with
...
```

If the name of a given is missing, the compiler will synthesize a name from the implemented type(s).

Note The name synthesized by the compiler is chosen to be readable and reasonably concise. For instance, the two instances above would get the names:

```
given_Ord_Int
given_Ord_List_T
```

The precise rules for synthesizing names are found here. These rules do not guarantee absence of name conflicts between given instances of types that are "too similar". To avoid conflicts one can use named instances.

Note To ensure robust binary compatibility, publicly available libraries should prefer named instances.

4.2.2 Alias Givens

An alias can be used to define a given instance that is equal to some expression. Example:

```
given global: ExecutionContext = ForkJoinPool()
```

This creates a given global of type ExecutionContext that resolves to the right hand side ForkJoinPool(). The first time global is accessed, a new ForkJoinPool is created, which is then returned for this and all subsequent accesses to global. This operation is thread-safe.

Alias givens can be anonymous as well, e.g.

```
given Position = enclosingTree.position
given (using config: Config): Factory = MemoizingFactory(config)
```

An alias given can have type parameters and context parameters just like any other given, but it can only implement a single type.

4.2.3 Given Macros

Given aliases can have the inline and transparent modifiers. Example:

```
transparent inline given mkAnnotations[A, T]: Annotations[A, T] = ${
   // code producing a value of a subtype of Annotations
}
```

Since mkAnnotations is transparent, the type of an application is the type of its right hand side, which can be a proper subtype of the declared result type Annotations[A, T].

4.2.4 Pattern-Bound Given Instances

Given instances can also appear in patterns. Example:

```
for given Context <- applicationContexts do
pair match
  case (ctx @ given Context, y) => ...
```

In the first fragment above, anonymous given instances for class Context are established by enumerating over applicationContexts. In the second fragment, a given Context instance named ctx is established by matching against the first half of the pair selector.

In each case, a pattern-bound given instance consists of given and a type T. The pattern matches exactly the same selectors as the type ascription pattern : T.

4.2.5 Negated Givens

Scala 2's somewhat puzzling behavior with respect to ambiguity has been exploited to implement the analogue of a "negated" search in implicit resolution, where a query Q1 fails if some other query Q2 succeeds and Q1 succeeds if Q2 fails. With the new cleaned up behavior these techniques no longer work. But the new special type scala.util.NotGiven now implements negation directly.

For any query type Q, NotGiven[Q] succeeds if and only if the implicit search for Q fails, for example:

```
import scala.util.NotGiven

trait Tagged[A]

case class Foo[A](value: Boolean)
object Foo:
    given fooTagged[A](using Tagged[A]): Foo[A] = Foo(true)
    given fooNotTagged[A](using NotGiven[Tagged[A]]): Foo[A] = Foo(false)

@main def test() =
    given Tagged[Int] with {}
    assert(summon[Foo[Int]].value) // fooTagged is found
    assert(!summon[Foo[String]].value) // fooNotTagged is found
```

4.2.6 Given Instance Initialization

A given instance without type or context parameters is initialized on-demand, the first time it is accessed. If a given has type or context parameters, a fresh instance is created for each reference.

4.2.7 Syntax

Here is the syntax for given instances:

 $TmplDef ::= \dots$

| 'given' GivenDef

GivenDef ::= [GivenSig] StructuralInstance

| [GivenSig] Type '=' Expr

| [GivenSig] Type

GivenSig ::= [id] [DefTypeParamClause] {UsingParamClause} ':'
StructuralInstance ::= ConstrApp {'with' ConstrApp} 'with' TemplateBody

A given instance starts with the reserved word given and an optional *signature*. The signature defines a name and/or parameters for the instance. It is followed by :. There are three kinds of given instances:

- A structural instance contains one or more types or constructor applications, followed by with and a template body that contains member definitions of the instance.
- An *alias instance* contains a type, followed by = and a right hand side expression.
- An abstract instance contains just the type, which is not followed by anything.

4.3 Using Clauses

Functional programming tends to express most dependencies as simple function parameterization. This is clean and powerful, but it sometimes leads to functions that take many parameters where the same value is passed over and over again in long call chains to many functions. Context parameters can help here since they enable the compiler to synthesize repetitive arguments instead of the programmer having to write them explicitly.

For example, with the given instances defined previously, a max function that works for any arguments for which an ordering exists can be defined as follows:

```
def max[T](x: T, y: T)(using ord: Ord[T]): T =
  if ord.compare(x, y) < 0 then y else x</pre>
```

Here, ord is a *context parameter* introduced with a using clause. The max function can be applied as follows:

```
max(2, 3)(using intOrd)
```

The (using intOrd) part passes intOrd as an argument for the ord parameter. But the point of context parameters is that this argument can also be left out (and it usually is). So the following applications are equally valid:

```
max(2, 3)
max(List(1, 2, 3), Nil)
```

4.3.1 Anonymous Context Parameters

In many situations, the name of a context parameter need not be mentioned explicitly at all, since it is used only in synthesized arguments for other context parameters. In that case one can avoid defining a parameter name and just provide its type. Example:

```
def maximum[T](xs: List[T])(using Ord[T]): T =
    xs.reduceLeft(max)
```

maximum takes a context parameter of type Ord only to pass it on as an inferred argument to max. The name of the parameter is left out.

Generally, context parameters may be defined either as a full parameter list (p_1: T_1, ..., p_n: T_n) or just as a sequence of types T_1, ..., T_n. Vararg parameters are not supported in using clauses.

4.3.2 Inferring Complex Arguments

Here are two other methods that have a context parameter of type Ord[T]:

```
def descending[T](using asc: Ord[T]): Ord[T] = new Ord[T]:
    def compare(x: T, y: T) = asc.compare(y, x)

def minimum[T](xs: List[T])(using Ord[T]) =
    maximum(xs)(using descending)
```

The minimum method's right hand side passes descending as an explicit argument to maximum(xs). With this setup, the following calls are all well-formed, and they all normalize to the last one:

```
minimum(xs)
maximum(xs)(using descending)
maximum(xs)(using descending(using listOrd))
maximum(xs)(using descending(using listOrd(using intOrd)))
```

4.3.3 Multiple using Clauses

There can be several using clauses in a definition and using clauses can be freely mixed with normal parameter clauses. Example:

```
def f(u: Universe)(using ctx: u.Context)(using s: ctx.Symbol, k: ctx.Kind) = ...
Multiple using clauses are matched left-to-right in applications. Example:
object global extends Universe { type Context = ... }
given ctx : global.Context with { type Symbol = ...; type Kind = ... }
given sym : ctx.Symbol
given kind: ctx.Kind
Then the following calls are all valid (and normalize to the last one)
f(global)
f(global) (using ctx)
f(global) (using ctx) (using sym, kind)
But f(global) (using sym, kind) would give a type error.
```

4.3.4 Summoning Instances

The method summon in Predef returns the given of a specific type. For example, the given instance for Ord[List[Int]] is produced by

```
summon[Ord[List[Int]]] // reduces to listOrd(using intOrd)
```

The **summon** method is simply defined as the (non-widening) identity function over a context parameter.

```
def summon[T](using x: T): x.type = x
```

4.3.5 Syntax

Here is the new syntax of parameters and arguments seen as a delta from the standard context free syntax of Scala 3. using is a soft keyword, recognized only at the start of a parameter or argument list. It can be used as a normal identifier everywhere else.

```
ClsParamClause ::= ... | UsingClsParamClause

DefParamClauses ::= ... | UsingParamClause

UsingClsParamClause ::= '(' 'using' (ClsParams | Types) ')'

UsingParamClause ::= '(' 'using' (DefParams | Types) ')'

ParArgumentExprs ::= ... | '(' 'using' ExprsInParens ')'
```

4.4 Context Bounds

A context bound is a shorthand for expressing the common pattern of a context parameter that depends on a type parameter. Using a context bound, the maximum function of the last section can be written like this:

```
def maximum[T: Ord](xs: List[T]): T = xs.reduceLeft(max)
```

A bound like: Ord on a type parameter T of a method or class indicates a context parameter with Ord[T]. The context parameter(s) generated from context bounds come last in the definition of the containing method or class. For instance,

```
def f[T: C1 : C2, U: C3](x: T)(using y: U, z: V): R
would expand to
```

```
\texttt{def f}[\texttt{T}, \texttt{U}](\texttt{x} \colon \texttt{T})(\texttt{using y} \colon \texttt{U}, \texttt{z} \colon \texttt{V})(\texttt{using C1}[\texttt{T}], \texttt{C2}[\texttt{T}], \texttt{C3}[\texttt{U}]) \colon \texttt{R}
```

Context bounds can be combined with subtype bounds. If both are present, subtype bounds come first, e.g.

```
def g[T <: B : C](x: T): R = ...
```

4.4.1 Migration

To ease migration, context bounds in Dotty map in Scala 3.0 to old-style implicit parameters for which arguments can be passed either with a (using ...) clause or with a normal application. From Scala 3.1 on, they will map to context parameters instead, as is described above.

If the source version is 3.1 and the -migration command-line option is set, any pairing of an evidence context parameter stemming from a context bound with a normal argument will give a migration warning. The warning indicates that a (using ...) clause is needed instead. The rewrite can be done automatically under -rewrite.

4.4.2 Syntax

```
TypeParamBounds ::= [SubtypeBounds] {ContextBound}
ContextBound ::= ':' Type
```

4.5 Importing Givens

A special form of import wildcard selector is used to import given instances. Example:

```
object A:
   class TC
   given tc: TC = ???
   def f(using TC) = ???
object B:
```

```
import A._
import A.given
```

In the code above, the import A._ clause in object B imports all members of A except the given instance tc. Conversely, the second import import A.given will import only that given instance. The two import clauses can also be merged into one:

```
object B:
   import A.{given, _}
```

Generally, a normal wildcard selector _ brings all definitions other than givens or extensions into scope whereas a given selector brings all givens (including those resulting from extensions) into scope.

There are two main benefits arising from these rules:

- It is made clearer where givens in scope are coming from. In particular, it is not possible to hide imported givens in a long list of regular wildcard imports.
- It enables importing all givens without importing anything else. This is particularly important since givens can be anonymous, so the usual recourse of using named imports is not practical.
- **4.5.0.1 Importing By Type** Since givens can be anonymous it is not always practical to import them by their name, and wildcard imports are typically used instead. By-type imports provide a more specific alternative to wildcard imports, which makes it clearer what is imported. Example:

```
import A.{given TC}
```

This imports any given in A that has a type which conforms to TC. Importing givens of several types T1,..., Tn is expressed by multiple given selectors.

```
import A.{given T1, ..., given Tn}
```

Importing all given instances of a parameterized type is expressed by wildcard arguments. For instance, assuming the object

object Instances:

```
given intOrd: Ordering[Int] = ...
given listOrd[T: Ordering]: Ordering[List[T]] = ...
given ec: ExecutionContext = ...
given im: Monoid[Int] = ...
```

the import clause

```
import Instances.{given Ordering[?], given ExecutionContext}
```

would import the intOrd, listOrd, and ec instances but leave out the im instance, since it fits none of the specified bounds.

By-type imports can be mixed with by-name imports. If both are present in an import clause, by-type imports come last. For instance, the import clause

```
import Instances.{im, given Ordering[?]}
```

would import im, intOrd, and listOrd but leave out ec.

4.5.0.2 Migration The rules for imports stated above have the consequence that a library would have to migrate in lockstep with all its users from old style implicits and normal imports to given and given imports.

The following modifications avoid this hurdle to migration.

- 1. A given import selector also brings old style implicits into scope. So, in Scala 3.0 an old-style implicit definition can be brought into scope either by a _ or a given wildcard selector.
- 2. In Scala 3.1, old-style implicits accessed through a _ wildcard import will give a deprecation warning.
- 3. In some version after 3.1, old-style implicits accessed through a _ wildcard import will give a compiler error.

These rules mean that library users can use given selectors to access old-style implicits in Scala 3.0, and will be gently nudged and then forced to do so in later versions. Libraries can then switch to given instances once their user base has migrated.

4.5.0.3 Syntax

```
'import' ImportExpr {',' ImportExpr}
Import
                  ::= StableId '.' ImportSpec
ImportExpr
ImportSpec
                  ::= id
                    1
                       'given'
                       '{' ImportSelectors) '}'
                      id ['=>' id | '=>' '_'] [',' ImportSelectors]
ImportSelectors
                  ::=
                      WildCardSelector {',' WildCardSelector}
                   ::=
WildCardSelector
                   'given' [InfixType]
Export
                      'export' ImportExpr {',' ImportExpr}
```

4.6 Extension Methods

Extension methods allow one to add methods to a type after the type is defined. Example:

```
case class Circle(x: Double, y: Double, radius: Double)
extension (c: Circle)
  def circumference: Double = c.radius * math.Pi * 2
```

Like regular methods, extension methods can be invoked with infix .:

```
val circle = Circle(0, 0, 1)
circle.circumference
```

4.6.0.1 Translation of Extension Methods An extension method translates to a specially labelled method that takes the leading parameter section as its first argument list. The label, expressed as **<extension>** here, is compiler-internal. So, the definition of **circumference** above translates to the following method, and can also be invoked as such:

```
<extension> def circumference(c: Circle): Double = c.radius * math.Pi * 2
assert(circle.circumference == circumference(circle))
```

4.6.0.2 Operators The extension method syntax can also be used to define operators. Examples:

```
extension (x: String)
  def < (y: String): Boolean = ...
extension (x: Elem)
  def +: (xs: Seq[Elem]): Seq[Elem] = ...
extension (x: Number)
  infix def min (y: Number): Number = ...

"ab" < "c"
1 +: List(2, 3)
x min 3

The three definitions above translate to

<extension> def < (x: String)(y: String): Boolean = ...
<extension> def +: (xs: Seq[Elem])(x: Elem): Seq[Elem] = ...
<extension> infix def min(x: Number)(y: Number): Number = ...
```

Note the swap of the two parameters x and xs when translating the right-associative operator +: to an extension method. This is analogous to the implementation of right binding operators as normal methods. The Scala compiler preprocesses an infix operation x +: xs to xs.+:(x), so the extension method ends up being applied to the sequence as first argument (in other words, the two swaps cancel each other out).

4.6.0.3 Generic Extensions It is also possible to extend generic types by adding type parameters to an extension. For instance:

```
extension [T](xs: List[T])
  def second = xs.tail.head
```

```
extension [T: Numeric](x: T)
  def + (y: T): T = summon[Numeric[T]].plus(x, y)
```

If an extension method has type parameters, they come immediately after extension and are followed by the extended parameter. When calling a generic extension method, any explicitly given type arguments follow the method name. So the second method could be instantiated as follows:

```
List(1, 2, 3).second[Int]
```

Of course, the type argument here would usually be left out since it can be inferred.

Extensions can also take using clauses. For instance, the + extension above could equivalently be written with a using clause:

```
extension [T](x: T)(using n: Numeric[T])
def + (y: T): T = n.plus(x, y)
```

Note: Type parameters have to be given after the extension keyword; they cannot be given after the def. This restriction might be lifted in the future once we support multiple type parameter clauses in a method. By contrast, using clauses can be defined for the extension as well as per def.

4.6.0.4 Collective Extensions Sometimes, one wants to define several extension methods that share the same left-hand parameter type. In this case one can "pull out" the common parameters into a single extension and enclose all methods in braces or an indented region. Example:

```
extension (ss: Seq[String])

def longestStrings: Seq[String] =
   val maxLength = ss.map(_.length).max
   ss.filter(_.length == maxLength)

def longestString: String = longestStrings.head
```

The same can be written with braces as follows (note that indented regions can still be used inside braces):

```
extension (ss: Seq[String]) {
  def longestStrings: Seq[String] = {
     val maxLength = ss.map(_.length).max
     ss.filter(_.length == maxLength)
  }
  def longestString: String = longestStrings.head
}
```

Note the right-hand side of longestString: it calls longestStrings directly, implicitly assuming the common extended value ss as receiver.

Collective extensions like these are a shorthand for individual extensions where each method is defined separately. For instance, the first extension above expands to:

```
extension (ss: Seq[String])
  def longestStrings: Seq[String] =
    val maxLength = ss.map(_.length).max
    ss.filter(_.length == maxLength)

extension (ss: Seq[String])
  def longestString: String = ss.longestStrings.head
```

Collective extensions also can take type parameters and have using clauses. Example:

```
extension [T](xs: List[T])(using Ordering[T])
  def smallest(n: Int): List[T] = xs.sorted.take(n)
  def smallestIndices(n: Int): List[Int] =
    val limit = smallest(n).max
    xs.zipWithIndex.collect { case (x, i) if x <= limit => i }
```

4.6.0.5 Translation of Calls to Extension Methods To convert a reference to an extension method, the compiler has to know about the extension method. We say in this case that the extension method is *applicable* at the point of reference. There are four possible ways for an extension method to be applicable:

- 1. The extension method is visible under a simple name, by being defined or inherited or imported in a scope enclosing the reference.
- 2. The extension method is a member of some given instance that is visible at the point of the reference.
- 3. The reference is of the form r.m and the extension method is defined in the implicit scope of the type of r.
- 4. The reference is of the form r.m and the extension method is defined in some given instance in the implicit scope of the type of r.

Here is an example for the first rule:

```
trait IntOps:
    extension (i: Int) def isZero: Boolean = i == 0

extension (i: Int) def safeMod(x: Int): Option[Int] =
    // extension method defined in same scope IntOps
    if x.isZero then None
    else Some(i % x)

object IntOpsEx extends IntOps:
    extension (i: Int) def safeDiv(x: Int): Option[Int] =
    // extension method brought into scope via inheritance from IntOps
    if x.isZero then None
    else Some(i / x)
```

```
trait SafeDiv:
  import IntOpsEx._ // brings safeDiv and safeMod into scope

extension (i: Int) def divide(d: Int): Option[(Int, Int)] =
  // extension methods imported and thus in scope
  (i.safeDiv(d), i.safeMod(d)) match
     case (Some(d), Some(r)) => Some((d, r))
     case _ => None
```

By the second rule, an extension method can be made available by defining a given instance containing it, like this:

```
given ops1: IntOps with {} // brings safeMod into scope
1.safeMod(2)
```

By the third and fourth rule, an extension method is available if it is in the implicit scope of the receiver type or in a given instance in that scope. Example:

```
class List[T]:
    ...
object List:
    ...
    extension [T](xs: List[List[T]])
        def flatten: List[T] = xs.foldLeft(Nil: List[T])(_ ++ _)

given [T: Ordering]: Ordering[List[T]] with
        extension (xs: List[T])
            def < (ys: List[T]): Boolean = ...
end List

// extension method available since it is in the implicit scope
// of List[List[Int]]
List(List(1, 2), List(3, 4)).flatten

// extension method available since it is in the given Ordering[List[T]],
// which is itself in the implicit scope of List[Int]
List(1, 2) < List(3)</pre>
```

The precise rules for resolving a selection to an extension method are as follows.

Assume a selection e.m[Ts] where m is not a member of e, where the type arguments [Ts] are optional, and where T is the expected type. The following two rewritings are tried in order:

- 1. The selection is rewritten to m[Ts] (e).
- 2. If the first rewriting does not typecheck with expected type T, and there is an extension method m in some eligible object o, the selection is rewritten to

o.m[Ts](e). An object o is eligible if

- o forms part of the implicit scope of T, or
- o is a given instance that is visible at the point of the application, or
- o is a given instance in the implicit scope of T.

This second rewriting is attempted at the time where the compiler also tries an implicit conversion from T to a type containing m. If there is more than one way of rewriting, an ambiguity error results.

An extension method can also be referenced using a simple identifier without a preceding expression. If an identifier g appears in the body of an extension method f and refers to an extension method g that is defined in the same collective extension

```
extension (x: T)

def f ... = ... g ...

def g ...
```

the identifier is rewritten to x.g. This is also the case if f and g are the same method. Example:

```
extension (s: String)
  def position(ch: Char, n: Int): Int =
    if n < s.length && s(n) != ch then position(ch, n + 1)
    else n</pre>
```

The recursive call position(ch, n + 1) expands to s.position(ch, n + 1) in this case. The whole extension method rewrites to

```
def position(s: String)(ch: Char, n: Int): Int =
  if n < s.length && s(n) != ch then position(s)(ch, n + 1)
  else n</pre>
```

4.6.0.6 Syntax Here are the syntax changes for extension methods and collective extensions relative to the current syntax.

extension is a soft keyword. It is recognized as a keyword only if it appears at the start of a statement and is followed by [or (. In all other cases it is treated as an identifier.

4.7 Implementing Type classes

A type class is an abstract, parameterized type that lets you add new behavior to any closed data type without using sub-typing. This can be useful in multiple use-cases, for example:

- expressing how a type you don't own (from the standard or 3rd-party library) conforms to such behavior
- expressing such a behavior for multiple types without involving sub-typing relationships (one extends another) between those types (see: ad hoc polymorphism for instance)

Therefore in Scala 3, type classes are just traits with one or more parameters whose implementations are not defined through the **extends** keyword, but by **given instances**. Here are some examples of common type classes:

4.7.0.1 Semigroups and monoids Here's the Monoid type class definition:

```
trait SemiGroup[T]:
    extension (x: T) def combine (y: T): T

trait Monoid[T] extends SemiGroup[T]:
    def unit: T
```

An implementation of this Monoid type class for the type String can be the following:

```
given Monoid[String] with
  extension (x: String) def combine (y: String): String = x.concat(y)
  def unit: String = ""
```

Whereas for the type Int one could write the following:

```
given Monoid[Int] with
  extension (x: Int) def combine (y: Int): Int = x + y
  def unit: Int = 0
```

This monoid can now be used as *context bound* in the following combineAll method:

```
def combineAll[T: Monoid](xs: List[T]): T =
    xs.foldLeft(summon[Monoid[T]].unit)(_.combine(_))
```

To get rid of the summon[...] we can define a Monoid object as follows:

```
object Monoid:
   def apply[T](using m: Monoid[T]) = m
```

Which would allow to re-write the combineAll method this way:

```
def combineAll[T: Monoid](xs: List[T]): T =
    xs.foldLeft(Monoid[T].unit)(_.combine(_))
```

4.7.0.2 Functors A Functor for a type provides the ability for its values to be "mapped over", i.e. apply a function that transforms inside a value while remembering its shape. For example, to modify every element of a collection without dropping or adding elements. We can represent all types that can be "mapped over" with F. It's a type constructor: the type of its values becomes concrete when provided a type argument. Therefore we write it F[_], hinting that the type F takes another type as argument. The definition of a generic Functor would thus be written as:

```
trait Functor[F[_]]:
   def map[A, B](x: F[A], f: A => B): F[B]
```

Which could read as follows: "A Functor for the type constructor $F[_]$ represents the ability to transform F[A] to F[B] through the application of function f with type $A \Rightarrow B$ ". We call the Functor definition here a *type class*. This way, we could define an instance of Functor for the List type:

```
given Functor[List] with
  def map[A, B](x: List[A], f: A => B): List[B] =
    x.map(f) // List already has a `map` method
```

With this given instance in scope, everywhere a Functor is expected, the compiler will accept a List to be used.

For instance, we may write such a testing method:

```
\label{eq:def-assertTransformation} $$ $ [F[_]: Functor, A, B] (expected: F[B], original: F[A], mapping) $$ $$ assert(expected == summon[Functor[F]].map(original, mapping)) $$
```

And use it this way, for example:

```
assertTransformation(List("a1", "b1"), List("a", "b"), elt => s"\$\{elt\}1")
```

That's a first step, but in practice we probably would like the map function to be a method directly accessible on the type F. So that we can call map directly on instances of F, and get rid of the summon[Functor[F]] part. As in the previous example of Monoids, extension methods help achieving that. Let's re-define the Functor type class with extension methods.

```
trait Functor[F[_]]:
    extension [A, B](x: F[A])
    def map(f: A => B): F[B]
```

The instance of Functor for List now becomes:

It simplifies the assertTransformation method:

```
def assertTransformation[F[_]: Functor, A, B](expected: F[B], original: F[A], mappi
   assert(expected == original.map(mapping))
```

The map method is now directly used on original. It is available as an extension method since original's type is F[A] and a given instance for Functor[F[A]] which defines map is in scope.

4.7.0.3 Monads Applying map in Functor[List] to a mapping function of type A => B results in a List[B]. So applying it to a mapping function of type A => List[B] results in a List[List[B]]. To avoid managing lists of lists, we may want to "flatten" the values in a single list.

That's where Monad comes in. A Monad for type F[_] is a Functor[F] with two more operations:

- flatMap, which turns an F[A] into an F[B] when given a function of type $A \Rightarrow F[B]$,
- pure, which creates an F[A] from a single value A.

Here is the translation of this definition in Scala 3:

```
trait Monad[F[_]] extends Functor[F]:

   /** The unit value for a monad */
   def pure[A](x: A): F[A]

   extension [A, B](x: F[A])
        /** The fundamental composition operation */
        def flatMap(f: A => F[B]): F[B]

        /** The `map` operation can now be defined in terms of `flatMap` */
        def map(f: A => B) = x.flatMap(f.andThen(pure))
```

end Monad

4.7.0.3.1 List A List can be turned into a monad via this given instance:

```
given listMonad: Monad[List] with
  def pure[A](x: A): List[A] =
     List(x)
  extension [A, B](xs: List[A])
   def flatMap(f: A => List[B]): List[B] =
     xs.flatMap(f) // rely on the existing `flatMap` method of `List`
```

Since Monad is a subtype of Functor, List is also a functor. The Functor's map operation is already provided by the Monad trait, so the instance does not need to define it explicitly.

4.7.0.3.2 Option Option is an other type having the same kind of behaviour:

```
given optionMonad: Monad[Option] with
  def pure[A](x: A): Option[A] =
```

```
Option(x)
extension [A, B](xo: Option[A])
  def flatMap(f: A => Option[B]): Option[B] = xo match
     case Some(x) => f(x)
     case None => None
```

4.7.0.3.3 Reader Another example of a Monad is the *Reader* Monad, which acts on functions instead of data types like List or Option. It can be used to combine multiple functions that all need the same parameter. For instance multiple functions needing access to some configuration, context, environment variables, etc.

Let's define a Config type, and two functions using it:

```
trait Config
// ...
def compute(i: Int)(config: Config): String = ???
def show(str: String)(config: Config): Unit = ???
```

We may want to combine compute and show into a single function, accepting a Config as parameter, and showing the result of the computation, and we'd like to use a monad to avoid passing the parameter explicitly multiple times. So postulating the right flatMap operation, we could write:

```
def computeAndShow(i: Int): Config => Unit = compute(i).flatMap(show)
instead of
show(compute(i)(config))(config)
```

Let's define this m then. First, we are going to define a type named ConfigDependent representing a function that when passed a Config produces a Result.

```
type ConfigDependent[Result] = Config => Result
```

The monad instance will look like this:

given configDependentMonad: Monad[ConfigDependent] with

```
def pure[A](x: A): ConfigDependent[A] =
    config => x

extension [A, B](x: ConfigDependent[A])
  def flatMap(f: A => ConfigDependent[B]): ConfigDependent[B] =
    config => f(x(config))(config)
```

end configDependentMonad

The type ConfigDependent can be written using type lambdas:

```
type ConfigDependent = [Result] =>> Config => Result
```

Using this syntax would turn the previous configDependentMonad into:

```
given configDependentMonad: Monad[[Result] =>> Config => Result] with

def pure[A](x: A): Config => A =
    config => x

extension [A, B](x: Config => A)
    def flatMap(f: A => Config => B): Config => B =
        config => f(x(config))(config)
```

end configDependentMonad

It is likely that we would like to use this pattern with other kinds of environments than our Config trait. The Reader monad allows us to abstract away Config as a type *parameter*, named Ctx in the following definition:

```
given readerMonad[Ctx]: Monad[[X] =>> Ctx => X] with

def pure[A](x: A): Ctx => A =
    ctx => x

extension [A, B](x: Ctx => A)
    def flatMap(f: A => Ctx => B): Ctx => B =
    ctx => f(x(ctx))(ctx)
```

end readerMonad

4.7.0.4 Summary The definition of a *type class* is expressed with a parameterised type with abstract members, such as a **trait**. The main difference between subtype polymorphism and ad-hoc polymorphism with *type classes* is how the definition of the *type class* is implemented, in relation to the type it acts upon. In the case of a *type class*, its implementation for a concrete type is expressed through a **given** instance definition, which is supplied as an implicit argument alongside the value it acts upon. With subtype polymorphism, the implementation is mixed into the parents of a class, and only a single term is required to perform a polymorphic operation. The type class solution takes more effort to set up, but is more extensible: Adding a new interface to a class requires changing the source code of that class. But contrast, instances for type classes can be defined anywhere.

To conclude, we have seen that traits and given instances, combined with other constructs like extension methods, context bounds and type lambdas allow a concise and natural expression of *type classes*.

4.8 Type Class Derivation

Type class derivation is a way to automatically generate given instances for type classes which satisfy some simple conditions. A type class in this sense is any trait or class with a type parameter determining the type being operated on. Common

examples are Eq. Ordering, or Show. For example, given the following Tree algebraic data type (ADT),

```
enum Tree[T] derives Eq, Ordering, Show:
   case Branch(left: Tree[T], right: Tree[T])
   case Leaf(elem: T)
```

The derives clause generates the following given instances for the Eq, Ordering and Show type classes in the companion object of Tree,

```
given [T: Eq] : Eq[Tree[T]] = Eq.derived
given [T: Ordering] : Ordering[Tree] = Ordering.derived
given [T: Show] : Show[Tree] = Show.derived
```

We say that Tree is the *deriving type* and that the Eq, Ordering and Show instances are *derived instances*.

4.8.0.1 Types supporting derives clauses All data types can have a derives clause. This document focuses primarily on data types which also have a given instance of the Mirror type class available. Instances of the Mirror type class are generated automatically by the compiler for,

- enums and enum cases
- case classes and case objects
- sealed classes or traits that have only case classes and case objects as children

Mirror type class instances provide information at the type level about the components and labelling of the type. They also provide minimal term level infrastructure to allow higher level libraries to provide comprehensive derivation support.

```
sealed trait Mirror:
```

```
/** the type being mirrored */
type MirroredType

/** the type of the elements of the mirrored type */
type MirroredElemTypes

/** The mirrored *-type */
type MirroredMonoType

/** The name of the type */
type MirroredLabel <: String

/** The names of the elements of the type */
type MirroredElemLabels <: Tuple

object Mirror:
/** The Mirror for a product type */</pre>
```

```
trait Product extends Mirror:

/** Create a new instance of type `T` with elements
  * taken from product `p`.
  */
  def fromProduct(p: scala.Product): MirroredMonoType

trait Sum extends Mirror:

/** The ordinal number of the case class of `x`.
  * For enums, `ordinal(x) == x.ordinal
  */
  def ordinal(x: MirroredMonoType): Int
```

end Mirror

Product types (i.e. case classes and objects, and enum cases) have mirrors which are subtypes of Mirror.Product. Sum types (i.e. sealed class or traits with product children, and enums) have mirrors which are subtypes of Mirror.Sum.

For the Tree ADT from above the following Mirror instances will be automatically provided by the compiler,

```
// Mirror for Tree
new Mirror.Sum:
   type MirroredType = Tree
   type MirroredElemTypes[T] = (Branch[T], Leaf[T])
   type MirroredMonoType = Tree[_]
   type MirroredLabels = "Tree"
   type MirroredElemLabels = ("Branch", "Leaf")
   def ordinal(x: MirroredMonoType): Int = x match
      case _: Branch[_] => 0
      case _: Leaf[_] => 1
// Mirror for Branch
new Mirror.Product:
   type MirroredType = Branch
   type MirroredElemTypes[T] = (Tree[T], Tree[T])
   type MirroredMonoType = Branch[_]
   type MirroredLabels = "Branch"
   type MirroredElemLabels = ("left", "right")
   def fromProduct(p: Product): MirroredMonoType =
      new Branch(...)
// Mirror for Leaf
new Mirror.Product:
```

```
type MirroredType = Leaf
type MirroredElemTypes[T] = Tuple1[T]
type MirroredMonoType = Leaf[_]
type MirroredLabels = "Leaf"
type MirroredElemLabels = Tuple1["elem"]

def fromProduct(p: Product): MirroredMonoType =
    new Leaf(...)
```

Note the following properties of Mirror types,

- Properties are encoded using types rather than terms. This means that they have no runtime footprint unless used and also that they are a compile time feature for use with Scala 3's metaprogramming facilities.
- The kinds of MirroredType and MirroredElemTypes match the kind of the data type the mirror is an instance for. This allows Mirrors to support ADTs of all kinds.
- There is no distinct representation type for sums or products (ie. there is no HList or Coproduct type as in Scala 2 versions of Shapeless). Instead the collection of child types of a data type is represented by an ordinary, possibly parameterized, tuple type. Scala 3's metaprogramming facilities can be used to work with these tuple types as-is, and higher level libraries can be built on top of them.
- For both product and sum types, the elements of MirroredElemTypes are arranged in definition order (i.e. Branch[T] precedes Leaf[T] in MirroredElemTypes for Tree because Branch is defined before Leaf in the source file). This means that Mirror.Sum differs in this respect from Shapeless's generic representation for ADTs in Scala 2, where the constructors are ordered alphabetically by name.
- The methods ordinal and fromProduct are defined in terms of MirroredMonoType which is the type of kind-* which is obtained from MirroredType by wildcarding its type parameters.

4.8.0.2 Type classes supporting automatic deriving A trait or class can appear in a derives clause if its companion object defines a method named derived. The signature and implementation of a derived method for a type class TC[_] are arbitrary but it is typically of the following form,

```
def derived[T](using Mirror.Of[T]): TC[T] = ...
```

That is, the derived method takes a context parameter of (some subtype of) type Mirror which defines the shape of the deriving type T, and computes the type class implementation according to that shape. This is all that the provider of an ADT with a derives clause has to know about the derivation of a type class instance.

Note that derived methods may have context Mirror parameters indirectly (e.g. by having a context argument which in turn has a context Mirror parameter, or not at all (e.g. they might use some completely different user-provided mechanism, for instance using Scala 3 macros or runtime reflection). We expect that (direct or

indirect) Mirror based implementations will be the most common and that is what this document emphasises.

Type class authors will most likely use higher level derivation or generic programming libraries to implement derived methods. An example of how a derived method might be implemented using *only* the low level facilities described above and Scala 3's general metaprogramming features is provided below. It is not anticipated that type class authors would normally implement a derived method in this way, however this walkthrough can be taken as a guide for authors of the higher level derivation libraries that we expect typical type class authors will use (for a fully worked out example of such a library, see Shapeless 3).

4.8.0.2.1 How to write a type class derived method using low level mechanisms The low-level method we will use to implement a type class derived method in this example exploits three new type-level constructs in Scala 3: inline methods, inline matches, and implicit searches via summonInline or summonFrom. Given this definition of the Eq type class,

```
trait Eq[T]:
   def eqv(x: T, y: T): Boolean
```

we need to implement a method Eq.derived on the companion object of Eq that produces a given instance for Eq[T] given a Mirror[T]. Here is a possible implementation,

Note that derived is defined as an inline given. This means that the method will be expanded at call sites (for instance the compiler generated instance definitions in the companion objects of ADTs which have a derived Eq clause), and also that it can be used recursively if necessary, to compute instances for children.

The body of this method (1) first materializes the Eq instances for all the child types of type the instance is being derived for. This is either all the branches of a sum type or all the fields of a product type. The implementation of summonAll is inline and uses Scala 3's summonInline construct to collect the instances as a List,

```
inline def summonAll[T <: Tuple]: List[Eq[_]] =
  inline erasedValue[T] match
    case _: EmptyTuple => Nil
    case : (t *: ts) => summonInline[Eq[t]] :: summonAll[ts]
```

with the instances for children in hand the derived method uses an inline match to dispatch to methods which can construct instances for either sums or products (2). Note that because derived is inline the match will be resolved at compile-time

and only the left-hand side of the matching case will be inlined into the generated code with types refined as revealed by the match.

In the sum case, eqSum, we use the runtime ordinal values of the arguments to eqv to first check if the two values are of the same subtype of the ADT (3) and then, if they are, to further test for equality based on the Eq instance for the appropriate ADT subtype using the auxiliary method check (4).

In the product case, eqProduct we test the runtime values of the arguments to eqv for equality as products based on the Eq instances for the fields of the data type (5),

```
def eqProduct[T](p: Mirror.ProductOf[T], elems: List[Eq[_]]): Eq[T] =
   new Eq[T]:
    def eqv(x: T, y: T): Boolean =
        iterator(x).zip(iterator(y)).zip(elems.iterator).forall { // (5)
            case ((x, y), elem) => check(elem)(x, y)
        }
```

Pulling this all together we have the following complete implementation,

```
import scala.deriving.
import scala.compiletime.{erasedValue, summonInline}
inline def summonAll[T <: Tuple]: List[Eq[_]] =
   inline erasedValue[T] match
      case : EmptyTuple => Nil
      case _: (t *: ts) => summonInline[Eq[t]] :: summonAll[ts]
trait Eq[T]:
   def eqv(x: T, y: T): Boolean
object Eq:
   given Eq[Int] with
      def eqv(x: Int, y: Int) = x == y
   def check(elem: Eq[_])(x: Any, y: Any): Boolean =
      elem.asInstanceOf[Eq[Any]].eqv(x, y)
   def iterator[T](p: T) = p.asInstanceOf[Product].productIterator
   def eqSum[T](s: Mirror.SumOf[T], elems: => List[Eq[_]]): Eq[T] =
      new Eq[T]:
         def eqv(x: T, y: T): Boolean =
```

```
val ordx = s.ordinal(x)
             (s.ordinal(y) == ordx) && check(elems(ordx))(x, y)
   def eqProduct[T](p: Mirror.ProductOf[T], elems: => List[Eq[]]): Eq[T] =
      new Eq[T]:
         def eqv(x: T, y: T): Boolean =
             iterator(x).zip(iterator(y)).zip(elems.iterator).forall {
                case ((x, y), elem) \Rightarrow check(elem)(x, y)
             }
   inline given derived[T](using m: Mirror.Of[T]): Eq[T] =
      lazy val elemInstances = summonAll[m.MirroredElemTypes]
      inline m match
         case s: Mirror.SumOf[T]
                                       => eqSum(s, elemInstances)
         case p: Mirror.ProductOf[T] => eqProduct(p, elemInstances)
end Eq
we can test this relative to a simple ADT like so,
enum Opt[+T] derives Eq:
   case Sm(t: T)
   case Nn
@main def test =
   import Opt.
   val eqoi = summon[Eq[Opt[Int]]]
   assert(eqoi.eqv(Sm(23), Sm(23)))
   assert(!eqoi.eqv(Sm(23), Sm(13)))
   assert(!eqoi.eqv(Sm(23), Nn))
In this case the code that is generated by the inline expansion for the derived Eq
instance for Opt looks like the following, after a little polishing,
given derived$Eq[T](using eqT: Eq[T]): Eq[Opt[T]] =
   eqSum(
      summon [Mirror [Opt [T]]],
      List(
         eqProduct(summon[Mirror[Sm[T]]], List(summon[Eq[T]])),
         eqProduct(summon[Mirror[Nn.type]], Nil)
      )
   )
```

Alternative approaches can be taken to the way that derived methods can be defined. For example, more aggressively inlined variants using Scala 3 macros, whilst being more involved for type class authors to write than the example above, can produce code for type classes like Eq which eliminate all the abstraction artefacts (eg. the Lists of child instances in the above) and generate code which is indistinguishable from what a programmer might write by hand. As a third example, using a higher level library such as Shapeless the type class author could define an

equivalent derived method as,

The framework described here enables all three of these approaches without mandating any of them.

For a brief discussion on how to use macros to write a type class derived method please read more at How to write a type class derived method using macros.

4.8.0.3 Deriving instances elsewhere Sometimes one would like to derive a type class instance for an ADT after the ADT is defined, without being able to change the code of the ADT itself. To do this, simply define an instance using the derived method of the type class as right-hand side. E.g, to implement Ordering for Option define,

```
given [T: Ordering]: Ordering[Option[T]] = Ordering.derived
```

Assuming the Ordering.derived method has a context parameter of type Mirror[T] it will be satisfied by the compiler generated Mirror instance for Option and the derivation of the instance will be expanded on the right hand side of this definition in the same way as an instance defined in ADT companion objects.

4.8.0.4 Syntax

Note: To align extends clauses and derives clauses, Scala 3 also allows multiple extended types to be separated by commas. So the following is now legal:

```
class A extends B, C { ... }
It is equivalent to the old form
class A extends B with C { ... }
```

4.8.0.5 Discussion This type class derivation framework is intentionally very small and low-level. There are essentially two pieces of infrastructure in compiler-generated Mirror instances,

- type members encoding properties of the mirrored types.
- a minimal value level mechanism for working generically with terms of the mirrored types.

The Mirror infrastructure can be seen as an extension of the existing Product infrastructure for case classes: typically Mirror types will be implemented by the ADTs companion object, hence the type members and the ordinal or fromProduct methods will be members of that object. The primary motivation for this design decision, and the decision to encode properties via types rather than terms was to keep the bytecode and runtime footprint of the feature small enough to make it possible to provide Mirror instances unconditionally.

Whilst Mirrors encode properties precisely via type members, the value level ordinal and fromProduct are somewhat weakly typed (because they are defined in terms of MirroredMonoType) just like the members of Product. This means that code for generic type classes has to ensure that type exploration and value selection proceed in lockstep and it has to assert this conformance in some places using casts. If generic type classes are correctly written these casts will never fail.

As mentioned, however, the compiler-provided mechanism is intentionally very low level and it is anticipated that higher level type class derivation and generic programming libraries will build on this and Scala 3's other metaprogramming facilities to hide these low-level details from type class authors and general users. Type class derivation in the style of both Shapeless and Magnolia are possible (a prototype of Shapeless 3, which combines aspects of both Shapeless 2 and Magnolia has been developed alongside this language feature) as is a more aggressively inlined style, supported by Scala 3's new quote/splice macro and inlining facilities.

4.9 Multiversal Equality

Previously, Scala had universal equality: Two values of any types could be compared with each other with == and !=. This came from the fact that == and != are implemented in terms of Java's equals method, which can also compare values of any two reference types.

Universal equality is convenient. But it is also dangerous since it undermines type safety. For instance, let's assume one is left after some refactoring with an erroneous program where a value y has type S instead of the correct type T.

```
val x = ... // of type T
val y = ... // of type S, but should be T
x == y // typechecks, will always yield false
```

If y gets compared to other values of type T, the program will still typecheck, since values of all types can be compared with each other. But it will probably give unexpected results and fail at runtime.

Multiversal equality is an opt-in way to make universal equality safer. It uses a binary type class CanEqual to indicate that values of two given types can be compared with each other. The example above would not typecheck if S or T was a class that derives CanEqual, e.g.

```
class T derives CanEqual
```

Alternatively, one can also provide a CanEqual given instance directly, like this:

```
given CanEqual[T, T] = CanEqual.derived
```

This definition effectively says that values of type T can (only) be compared to other values of type T when using == or !=. The definition affects type checking but it has no significance for runtime behavior, since == always maps to equals and != always maps to the negation of equals. The right hand side CanEqual.derived of the definition is a value that has any CanEqual instance as its type. Here is the definition of class CanEqual and its companion object:

```
package scala
import annotation.implicitNotFound
```

```
@implicitNotFound("Values of types ${L} and ${R} cannot be compared with == or !=")
sealed trait CanEqual[-L, -R]
```

```
object CanEqual:
```

```
object derived extends CanEqual[Any, Any]
```

One can have several CanEqual given instances for a type. For example, the four definitions below make values of type A and type B comparable with each other, but not comparable to anything else:

```
given CanEqual[A, A] = CanEqual.derived
given CanEqual[B, B] = CanEqual.derived
given CanEqual[A, B] = CanEqual.derived
given CanEqual[B, A] = CanEqual.derived
```

The scala.CanEqual object defines a number of CanEqual given instances that together define a rule book for what standard types can be compared (more details below).

There is also a "fallback" instance named canEqualAny that allows comparisons over all types that do not themselves have a CanEqual given. canEqualAny is defined as follows:

```
def canEqualAny[L, R]: CanEqual[L, R] = CanEqual.derived
```

Even though canEqualAny is not declared as given, the compiler will still construct an canEqualAny instance as answer to an implicit search for the type CanEqual[L, R], unless L or R have CanEqual instances defined on them, or the language feature strictEquality is enabled.

The primary motivation for having canEqualAny is backwards compatibility. If

this is of no concern, one can disable canEqualAny by enabling the language feature strictEquality. As for all language features this can be either done with an import

```
import scala.language.strictEquality
```

or with a command line option -language:strictEquality.

4.9.1 Deriving CanEqual Instances

Instead of defining CanEqual instances directly, it is often more convenient to derive them. Example:

```
class Box[T](x: T) derives CanEqual
```

By the usual rules of type class derivation, this generates the following CanEqual instance in the companion object of Box:

```
given [T, U](using CanEqual[T, U]): CanEqual[Box[T], Box[U]] =
   CanEqual.derived
```

That is, two boxes are comparable with == or != if their elements are. Examples:

```
new Box(1) == new Box(1L) // ok since there is an instance for `CanEqual[Int, Lonew Box(1) == new Box("a") // error: can't compare new Box(1) == 1 // error: can't compare
```

4.9.2 Precise Rules for Equality Checking

The precise rules for equality checking are as follows.

If the strictEquality feature is enabled then a comparison using x == y or x != y between values x: T and y: U is legal if there is a given of type CanEqual[T, U].

In the default case where the **strictEquality** feature is not enabled the comparison is also legal if

- 1. T and U are the same, or
- 2. one of T, U is a subtype of the *lifted* version of the other type, or
- 3. neither T nor U have a reflexive CanEqual instance.

Explanations:

- *lifting* a type S means replacing all references to abstract types in covariant positions of S by their upper bound, and replacing all refinement types in covariant positions of S by their parent.
- a type T has a *reflexive* CanEqual instance if the implicit search for CanEqual[T, T] succeeds.

4.9.3 Predefined CanEqual Instances

The CanEqual object defines instances for comparing - the primitive types Byte, Short, Char, Int, Long, Float, Double, Boolean, and Unit, - java.lang.Number,

java.lang.Boolean, and java.lang.Character, - scala.collection.Seq, and scala.collection.Set.

Instances are defined so that every one of these types has a *reflexive* CanEqual instance, and the following holds:

- Primitive numeric types can be compared with each other.
- Primitive numeric types can be compared with subtypes of java.lang.Number (and *vice versa*).
- Boolean can be compared with java.lang.Boolean (and *vice versa*).
- Char can be compared with java.lang.Character (and *vice versa*).
- Two sequences (of arbitrary subtypes of scala.collection.Seq) can be compared with each other if their element types can be compared. The two sequence types need not be the same.
- Two sets (of arbitrary subtypes of scala.collection.Set) can be compared with each other if their element types can be compared. The two set types need not be the same.
- Any subtype of AnyRef can be compared with Null (and vice versa).

4.9.4 Why Two Type Parameters?

One particular feature of the CanEqual type is that it takes *two* type parameters, representing the types of the two items to be compared. By contrast, conventional implementations of an equality type class take only a single type parameter which represents the common type of *both* operands. One type parameter is simpler than two, so why go through the additional complication? The reason has to do with the fact that, rather than coming up with a type class where no operation existed before, we are dealing with a refinement of pre-existing, universal equality. It is best illustrated through an example.

Say you want to come up with a safe version of the contains method on List[T]. The original definition of contains in the standard library was:

```
class List[+T]:
    ...
    def contains(x: Any): Boolean
```

That uses universal equality in an unsafe way since it permits arguments of any type to be compared with the list's elements. The "obvious" alternative definition

```
def contains(x: T): Boolean
```

does not work, since it refers to the covariant parameter T in a nonvariant context. The only variance-correct way to use the type parameter T in contains is as a lower bound:

```
def contains[U >: T](x: U): Boolean
```

This generic version of contains is the one used in the current (Scala 2.13) version of List. It looks different but it admits exactly the same applications as the

contains(x: Any) definition we started with. However, we can make it more useful (i.e. restrictive) by adding a CanEqual parameter:

```
def contains[U >: T](x: U)(using CanEqual[T, U]): Boolean // (1)
```

This version of contains is equality-safe! More precisely, given x: T, xs: List[T] and y: U, then xs.contains(y) is type-correct if and only if x == y is type-correct.

Unfortunately, the crucial ability to "lift" equality type checking from simple equality and pattern matching to arbitrary user-defined operations gets lost if we restrict ourselves to an equality class with a single type parameter. Consider the following signature of contains with a hypothetical CanEqual1[T] type class:

```
def contains[U >: T](x: U)(using CanEqual1[U]): Boolean // (2)
```

This version could be applied just as widely as the original contains(x: Any) method, since the CanEqual1[Any] fallback is always available! So we have gained nothing. What got lost in the transition to a single parameter type class was the original rule that CanEqual[A, B] is available only if neither A nor B have a reflexive CanEqual instance. That rule simply cannot be expressed if there is a single type parameter for CanEqual.

The situation is different under -language:strictEquality. In that case, the CanEqual[Any, Any] or CanEqual1[Any] instances would never be available, and the single and two-parameter versions would indeed coincide for most practical purposes.

But assuming -language:strictEquality immediately and everywhere poses migration problems which might well be unsurmountable. Consider again contains, which is in the standard library. Parameterizing it with the CanEqual type class as in (1) is an immediate win since it rules out non-sensical applications while still allowing all sensible ones. So it can be done almost at any time, modulo binary compatibility concerns. On the other hand, parameterizing contains with CanEqual1 as in (2) would make contains unusable for all types that have not yet declared a CanEqual1 instance, including all types coming from Java. This is clearly unacceptable. It would lead to a situation where, rather than migrating existing libraries to use safe equality, the only upgrade path is to have parallel libraries, with the new version only catering to types deriving CanEqual1 and the old version dealing with everything else. Such a split of the ecosystem would be very problematic, which means the cure is likely to be worse than the disease.

For these reasons, it looks like a two-parameter type class is the only way forward because it can take the existing ecosystem where it is and migrate it towards a future where more and more code uses safe equality.

In applications where -language:strictEquality is the default one could also introduce a one-parameter type alias such as

```
type Eq[-T] = CanEqual[T, T]
```

Operations needing safe equality could then use this alias instead of the two-parameter CanEqual class. But it would only work under -language:strictEquality,

since otherwise the universal Eq[Any] instance would be available everywhere.

More on multiversal equality is found in a blog post and a GitHub issue.

4.10 Context Functions

Context functions are functions with (only) context parameters. Their types are context function types. Here is an example of a context function type:

```
type Executable[T] = ExecutionContext ?=> T
```

Context functions are written using ?=> as the "arrow" sign. They are applied to synthesized arguments, in the same way methods with context parameters are applied. For instance:

```
given ec: ExecutionContext = ...

def f(x: Int): ExecutionContext ?=> Int = ...

// could be written as follows with the type alias from above

// def f(x: Int): Executable[Int] = ...

f(2)(using ec) // explicit argument
f(2) // argument is inferred
```

Conversely, if the expected type of an expression E is a context function type (T_1, ..., T_n) ?=> U and E is not already an context function literal, E is converted to a context function literal by rewriting it to

```
(x 1: T1, \ldots, x n: Tn) ?=> E
```

where the names $x_1, ..., x_n$ are arbitrary. This expansion is performed before the expression E is typechecked, which means that $x_1, ..., x_n$ are available as givens in E.

Like their types, context function literals are written using ?=> as the arrow between parameters and results. They differ from normal function literals in that their types are context function types.

For example, continuing with the previous definitions,

```
def g(arg: Executable[Int]) = ... g(22) \hspace{1cm} // \hspace{1cm} is \hspace{1cm} expanded \hspace{1cm} to \hspace{1cm} g((ev: ExecutionContext) ?=> 22) g(f(2)) \hspace{1cm} // \hspace{1cm} is \hspace{1cm} expanded \hspace{1cm} to \hspace{1cm} g((ev: ExecutionContext) ?=> f(2) (using \hspace{1cm} ev)) g((ctx: ExecutionContext) ?=> f(3)) \hspace{1cm} // \hspace{1cm} is \hspace{1cm} expanded \hspace{1cm} to \hspace{1cm} g((ctx: ExecutionContext) ?=> f(3) (using \hspace{1cm} ctx)) \hspace{1cm} // \hspace{1cm} is \hspace{1cm} left \hspace{1cm} as \hspace{1cm} it \hspace{1cm} left \hspace{1cm} l
```

4.10.0.1 Example: Builder Pattern Context function types have considerable expressive power. For instance, here is how they can support the "builder pattern", where the aim is to construct tables like this:

```
table {
   row {
      cell("top left")
      cell("top right")
   }
  row {
      cell("bottom left")
      cell("bottom right")
   }
}
```

The idea is to define classes for Table and Row that allow the addition of elements via add:

```
class Table:
    val rows = new ArrayBuffer[Row]
    def add(r: Row): Unit = rows += r
    override def toString = rows.mkString("Table(", ", ", ")")

class Row:
    val cells = new ArrayBuffer[Cell]
    def add(c: Cell): Unit = cells += c
    override def toString = cells.mkString("Row(", ", ", ")")

case class Cell(elem: String)
```

Then, the table, row and cell constructor methods can be defined with context function types as parameters to avoid the plumbing boilerplate that would otherwise be necessary.

```
def table(init: Table ?=> Unit) =
   given t: Table = Table()
   init
   t

def row(init: Row ?=> Unit)(using t: Table) =
   given r: Row = Row()
   init
   t.add(r)

def cell(str: String)(using r: Row) =
   r.add(new Cell(str))
```

With that setup, the table construction code above compiles and expands to:

```
table { ($t: Table) ?=>
```

```
row { ($r: Row) ?=>
    cell("top left")(using $r)
    cell("top right")(using $r)
}(using $t)

row { ($r: Row) ?=>
    cell("bottom left")(using $r)
    cell("bottom right")(using $r)
}(using $t)
}
```

4.10.0.2 Example: Postconditions As a larger example, here is a way to define constructs for checking arbitrary postconditions using an extension method ensuring so that the checked result can be referred to simply by result. The example combines opaque type aliases, context function types, and extension methods to provide a zero-overhead abstraction.

```
object PostConditions:
    opaque type WrappedResult[T] = T

def result[T](using r: WrappedResult[T]): T = r

    extension [T](x: T)
        def ensuring(condition: WrappedResult[T] ?=> Boolean): T =
            assert(condition(using x))
            x
end PostConditions
import PostConditions.{ensuring, result}

val s = List(1, 2, 3).sum.ensuring(result == 6)
```

Explanations: We use a context function type WrappedResult[T] ?=> Boolean as the type of the condition of ensuring. An argument to ensuring such as (result == 6) will therefore have a given of type WrappedResult[T] in scope to pass along to the result method. WrappedResult is a fresh type, to make sure that we do not get unwanted givens in scope (this is good practice in all cases where context parameters are involved). Since WrappedResult is an opaque type alias, its values need not be boxed, and since ensuring is added as an extension method, its argument does not need boxing either. Hence, the implementation of ensuring is as about as efficient as the best possible code one could write by hand:

```
val s =
  val result = List(1, 2, 3).sum
  assert(result == 6)
  result
```

4.10.0.3 Reference For more information, see the blog article, (which uses a different syntax that has been superseded).

More details

4.11 Implicit Conversions

Implicit conversions are defined by given instances of the scala. Conversion class. This class is defined in package scala as follows:

```
abstract class Conversion[-T, +U] extends (T => U):
   def apply (x: T): U
```

For example, here is an implicit conversion from String to Token:

```
given Conversion[String, Token] with
  def apply(str: String): Token = new KeyWord(str)
```

Using an alias this can be expressed more concisely as:

```
given Conversion[String, Token] = new KeyWord(_)
```

An implicit conversion is applied automatically by the compiler in three situations:

- 1. If an expression **e** has type T, and T does not conform to the expression's expected type S.
- 2. In a selection e.m with e of type T, but T defines no member m.
- 3. In an application e.m(args) with e of type T, if T does define some member(s) named m, but none of these members can be applied to the arguments args.

In the first case, the compiler looks for a given scala. Conversion instance that maps an argument of type T to type S. In the second and third case, it looks for a given scala. Conversion instance that maps an argument of type T to a type that defines a member m which can be applied to args if present. If such an instance C is found, the expression e is replaced by C.apply(e).

4.11.1 Examples

1. The Predef package contains "auto-boxing" conversions that map primitive number types to subclasses of java.lang.Number. For instance, the conversion from Int to java.lang.Integer can be defined as follows:

```
given int2Integer: Conversion[Int, java.lang.Integer] =
   java.lang.Integer.valueOf(_)
```

2. The "magnet" pattern is sometimes used to express many variants of a method. Instead of defining overloaded versions of the method, one can also let the method take one or more arguments of specially defined "magnet" types, into which various argument types can be converted. Example:

```
object Completions:
```

```
// The argument "magnet" type
```

```
enum CompletionArg:
      case Error(s: String)
      case Response(f: Future[HttpResponse])
      case Status(code: Future[StatusCode])
  object CompletionArg:
   // conversions defining the possible arguments to pass to `complete`
   // these always come with CompletionArg
   // They can be invoked explicitly, e.g.
         CompletionArg.fromStatusCode(statusCode)
                         : Conversion[String, CompletionArg]
     given fromString
      given fromFuture : Conversion[Future[HttpResponse], CompletionArg] =
     given fromStatusCode: Conversion[Future[StatusCode], CompletionArg]
  end CompletionArg
   import CompletionArg._
  def complete[T](arg: CompletionArg) = arg match
      case Error(s) => ...
      case Response(f) => ...
      case Status(code) => ...
end Completions
```

This setup is more complicated than simple overloading of complete, but it can still be useful if normal overloading is not available (as in the case above, since we cannot have two overloaded methods that take Future[...] arguments), or if normal overloading would lead to a combinatorial explosion of variants.

4.12 By-Name Context Parameters

Context parameters can be declared by-name to avoid a divergent inferred expansion. Example:

```
trait Codec[T]:
    def write(x: T): Unit

given intCodec: Codec[Int] = ???

given optionCodec[T](using ev: => Codec[T]): Codec[Option[T]] with
    def write(xo: Option[T]) = xo match
        case Some(x) => ev.write(x)
        case None =>

val s = summon[Codec[Option[Int]]]
```

```
s.write(Some(33))
s.write(None)
```

As is the case for a normal by-name parameter, the argument for the context parameter ev is evaluated on demand. In the example above, if the option value x is None, it is not evaluated at all.

The synthesized argument for a context parameter is backed by a local val if this is necessary to prevent an otherwise diverging expansion.

The precise steps for synthesizing an argument for a by-name context parameter of type => T are as follows.

1. Create a new given of type T:

```
given lv: T = ???
```

where lv is an arbitrary fresh name.

- 2. This given is not immediately available as candidate for argument inference (making it immediately available could result in a loop in the synthesized computation). But it becomes available in all nested contexts that look again for an argument to a by-name context parameter.
- 3. If this search succeeds with expression E, and E contains references to lv, replace E by

```
{ given lv: T = E; lv }
```

Otherwise, return E unchanged.

In the example above, the definition of s would be expanded as follows.

```
val s = summon[Test.Codec[Option[Int]]](
   optionCodec[Int](using intCodec)
)
```

No local given instance was generated because the synthesized argument is not recursive.

4.12.0.1 Reference For more information, see Issue #1998 and the associated Scala SIP.

4.13 Relationship with Scala 2 Implicits 🗹

Many, but not all, of the new contextual abstraction features in Scala 3 can be mapped to Scala 2's implicits. This page gives a rundown on the relationships between new and old features.

4.13.1 Simulating Scala 3 Contextual Abstraction Concepts with Scala 2 Implicits

- **4.13.1.1** Given Instances Given instances can be mapped to combinations of implicit objects, classes and implicit methods.
 - 1. Given instances without parameters are mapped to implicit objects. For instance,

```
given intOrd: Ord[Int] with { ... }
maps to
implicit object intOrd extends Ord[Int] { ... }
```

2. Parameterized givens are mapped to combinations of classes and implicit methods. For instance,

```
given listOrd[T](using ord: Ord[T]): Ord[List[T]] with { ... }
maps to
class listOrd[T](implicit ord: Ord[T]) extends Ord[List[T]] { ... }
final implicit def listOrd[T](implicit ord: Ord[T]): listOrd[T] =
    new listOrd[T]
```

3. Alias givens map to implicit methods or implicit lazy vals. If an alias has neither type nor context parameters, it is treated as a lazy val, unless the right hand side is a simple reference, in which case we can use a forwarder to that reference without caching it.

Examples:

```
given global: ExecutionContext = new ForkJoinContext()

val ctx: Context
given Context = ctx

would map to

final implicit lazy val global: ExecutionContext = new ForkJoinContext()
final implicit def given Context = ctx
```

4.13.1.2 Anonymous Given Instances Anonymous given instances get compiler synthesized names, which are generated in a reproducible way from the implemented type(s). For example, if the names of the IntOrd and ListOrd givens above were left out, the following names would be synthesized instead:

```
given given_Ord_Int: Ord[Int] with { ... }
given given_Ord_List_T[T](using ord: Ord[T]): Ord[List[T]] with { ... }
```

The synthesized type names are formed from

- 1. the prefix given,
- 2. the simple name(s) of the implemented type(s), leaving out any prefixes,

3. the simple name(s) of the top-level argument type constructors to these types.

Tuples are treated as transparent, i.e. a type F[(X, Y)] would get the synthesized name F_X_Y . Directly implemented function types $A \Rightarrow B$ are represented as $A_{to}B$. Function types used as arguments to other type constructors are represented as Function.

4.13.1.3 Using Clauses Using clauses correspond largely to Scala 2's implicit parameter clauses. E.g.

```
def max[T](x: T, y: T)(using ord: Ord[T]): T
would be written
def max[T](x: T, y: T)(implicit ord: Ord[T]): T
```

in Scala 2. The main difference concerns applications of such parameters. Explicit arguments to parameters of using clauses *must* be written using (using ...), mirroring the definition syntax. E.g, max(2, 3)(using IntOrd). Scala 2 uses normal applications max(2, 3)(IntOrd) instead. The Scala 2 syntax has some inherent ambiguities and restrictions which are overcome by the new syntax. For instance, multiple implicit parameter lists are not available in the old syntax, even though they can be simulated using auxiliary objects in the "Aux" pattern.

The summon method corresponds to implicitly in Scala 2. It is precisely the same as the the method in Shapeless. The difference between summon (or the) and implicitly is that summon can return a more precise type than the type that was asked for.

4.13.1.4 Context Bounds Context bounds are the same in both language versions. They expand to the respective forms of implicit parameters.

Note: To ease migration, context bounds in Scala 3 map for a limited time to old-style implicit parameters for which arguments can be passed either in a using clause or in a normal argument list. Once old-style implicits are deprecated, context bounds will map to using clauses instead.

4.13.1.5 Extension Methods Extension methods have no direct counterpart in Scala 2, but they can be simulated with implicit classes. For instance, the extension method

```
extension (c: Circle)
  def circumference: Double = c.radius * math.Pi * 2

could be simulated to some degree by

implicit class CircleDecorator(c: Circle) extends AnyVal {
  def circumference: Double = c.radius * math.Pi * 2
}
```

Abstract extension methods in traits that are implemented in given instances have no direct counterpart in Scala 2. The only way to simulate these is to make implicit classes available through imports. The Simulacrum macro library can automate this process in some cases.

- **4.13.1.6** Type Class Derivation Type class derivation has no direct counterpart in the Scala 2 language. Comparable functionality can be achieved by macrobased libraries such as Shapeless, Magnolia, or scalaz-deriving.
- **4.13.1.7** Context Function Types Context function types have no analogue in Scala 2.
- **4.13.1.8 Implicit By-Name Parameters** Implicit by-name parameters are not supported in Scala 2, but can be emulated to some degree by the Lazy type in Shapeless.
- 4.13.2 Simulating Scala 2 Implicits in Scala 3
- **4.13.2.1 Implicit Conversions** Implicit conversion methods in Scala 2 can be expressed as given instances of the scala.Conversion class in Scala 3. For instance, instead of

```
implicit def stringToToken(str: String): Token = new Keyword(str)
one can write
given stringToToken: Conversion[String, Token] with
   def apply(str: String): Token = KeyWord(str)
or
given stringToToken: Conversion[String, Token] = KeyWord(_)
```

- **4.13.2.2** Implicit Classes Implicit classes in Scala 2 are often used to define extension methods, which are directly supported in Scala 3. Other uses of implicit classes can be simulated by a pair of a regular class and a given Conversion instance.
- **4.13.2.3** Implicit Values Implicit val definitions in Scala 2 can be expressed in Scala 3 using a regular val definition and an alias given. For instance, Scala 2's

```
lazy implicit val pos: Position = tree.sourcePos
can be expressed in Scala 3 as
lazy val pos: Position = tree.sourcePos
given Position = pos
```

4.13.2.4 Abstract Implicits An abstract implicit val or def in Scala 2 can be expressed in Scala 3 using a regular abstract definition and an alias given. For instance, Scala 2's

```
implicit def symDecorator: SymDecorator
can be expressed in Scala 3 as
def symDecorator: SymDecorator
given SymDecorator = symDecorator
```

4.13.3 Implementation Status and Timeline

The Scala 3 implementation implements both Scala 2's implicits and the new abstractions. In fact, support for Scala 2's implicits is an essential part of the common language subset between 2.13 and Scala 3. Migration to the new abstractions will be supported by making automatic rewritings available.

Depending on adoption patterns, old style implicits might start to be deprecated in a version following Scala 3.0.

4.14 How to write a type class derived method using macros

In the main derivation documentation page, we explained the details behind Mirrors and type class derivation. Here we demonstrate how to implement a type class derived method using macros only. We follow the same example of deriving Eq instances and for simplicity we support a Product type e.g., a case class Person. The low-level method we will use to implement the derived method exploits quotes, splices of both expressions and types and the scala.quoted.Expr.summon method which is the equivalent of summonFrom. The former is suitable for use in a quote context, used within macros.

As in the original code, the type class definition is the same:

```
trait Eq[T]:
   def eqv(x: T, y: T): Boolean
```

we need to implement a method Eq.derived on the companion object of Eq that produces a quoted instance for Eq[T]. Here is a possible signature,

```
\label{eq:given_derived} \mbox{given derived[T: Type] (using Quotes): } \mbox{Expr}[\mbox{Eq}[\mbox{T}]]
```

and for comparison reasons we give the same signature we had with inline:

```
inline given derived[T]: (m: Mirror.Of[T]) => Eq[T] = ???
```

Note, that since a type is used in a subsequent stage it will need to be lifted to a Type by using the corresponding context bound. Also, not that we can summon the quoted Mirror inside the body of the derived this we can omit it from the signature. The body of the derived method is shown below:

```
given derived[T: Type](using Quotes): Expr[Eq[T]] =
  import quotes.reflect._
```

```
val ev: Expr[Mirror.Of[T]] = Expr.summon[Mirror.Of[T]].get

ev match
    case '{ $m: Mirror.ProductOf[T] { type MirroredElemTypes = elementTypes }} =>
        val elemInstances = summonAll[elementTypes]
        val eqProductBody: (Expr[T], Expr[T]) => Expr[Boolean] = (x, y) =>
            elemInstances.zipWithIndex.foldLeft(Expr(true: Boolean)) {
            case (acc, (elem, index)) =>
                val e1 = '{$x.asInstanceOf[Product].productElement(${Expr(index)})
                val e2 = '{$y.asInstanceOf[Product].productElement(${Expr(index)})
                '{ $acc && $elem.asInstanceOf[Eq[Any]].eqv($e1, $e2) }
            }

            '{ eqProduct((x: T, y: T) => ${eqProductBody('x, 'y)}) }

// case for Mirror.ProductOf[T]
// ...
```

Note, that in the inline case we can merely write summonAll[m.MirroredElemTypes] inside the inline method but here, since Expr.summon is required, we can extract the element types in a macro fashion. Being inside a macro, our first reaction would be to write the code below. Since the path inside the type argument is not stable this cannot be used:

```
'{
    summonAll[$m.MirroredElemTypes]
}
```

Instead we extract the tuple-type for element types using pattern matching over quotes and more specifically of the refined type:

Shown below is the implementation of **summonAll** as a macro. We assume that given instances for our primitive types exist.

```
def summonAll[T: Type](using Quotes): List[Expr[Eq[_]]] =
   Type.of[T] match
    case '[String *: tpes] => '{ summon[Eq[String]] } :: summonAll[tpes]
    case '[Int *: tpes] => '{ summon[Eq[Int]] } :: summonAll[tpes]
    case '[tpe *: tpes] => derived[tpe] :: summonAll[tpes]
    case '[EmptyTuple] => Nil
```

One additional difference with the body of derived here as opposed to the one with inline is that with macros we need to synthesize the body of the code during the macro-expansion time. That is the rationale behind the eqProductBody function. Assuming that we calculate the equality of two Persons defined with a case class that holds a name of type String and an age of type Int, the equality check we want to generate is the following:

```
true
&& Eq[String].eqv(x.productElement(0),y.productElement(0))
&& Eq[Int].eqv(x.productElement(1), y.productElement(1))
```

4.14.1 Calling the derived method inside the macro

Following the rules in Macros we create two methods. One that hosts the top-level splice eqv and one that is the implementation. Alternatively and what is shown below is that we can call the eqv method directly. The eqGen can trigger the derivation.

```
extension [T] (inline x: T)
   inline def === (inline y: T)(using eq: Eq[T]): Boolean = eq.eqv(x, y)
inline given eqGen[T]: Eq[T] = ${ Eq.derived[T] }
Note, that we use inline method syntax and we can compare instance such as
Sm(Person("Test", 23)) === Sm(Person("Test", 24)) for e.g., the following
two types:
case class Person(name: String, age: Int)
enum Opt[+T]:
   case Sm(t: T)
   case Nn
The full code is shown below:
import scala.deriving.
import scala.quoted._
trait Eq[T]:
   def eqv(x: T, y: T): Boolean
object Eq:
   given Eq[String] with
      def eqv(x: String, y: String) = x == y
   given Eq[Int] with
      def eqv(x: Int, y: Int) = x == y
   def eqProduct[T](body: (T, T) => Boolean): Eq[T] =
      new Eq[T]:
         def eqv(x: T, y: T): Boolean = body(x, y)
   def eqSum[T](body: (T, T) => Boolean): Eq[T] =
      new Eq[T]:
         def eqv(x: T, y: T): Boolean = body(x, y)
```

```
def summonAll[T: Type](using Quotes): List[Expr[Eq[_]]] =
      Type.of[T] match
         case '[String *: tpes] => '{ summon[Eq[String]] } :: summonAll[tpes]
         case '[Int *: tpes] => '{ summon[Eq[Int]] } :: summonAll[tpes]
                                => derived[tpe] :: summonAll[tpes]
         case '[tpe *: tpes]
         case '[EmptyTuple]
                                => Nil
  given derived[T: Type] (using q: Quotes): Expr[Eq[T]] =
      import quotes.reflect.
     val ev: Expr[Mirror.Of[T]] = Expr.summon[Mirror.Of[T]].get
     ev match
         case '{ $m: Mirror.ProductOf[T] { type MirroredElemTypes = elementTypes }}
            val elemInstances = summonAll[elementTypes]
            val eqProductBody: (Expr[T], Expr[T]) => Expr[Boolean] = (x, y) =>
               elemInstances.zipWithIndex.foldLeft(Expr(true: Boolean)) {
                  case (acc, (elem, index)) =>
                     val e1 = '{$x.asInstanceOf[Product].productElement(${Expr(index)})
                     val e2 = '{$y.asInstanceOf[Product].productElement(${Expr(index)}
                     '{ $acc && $elem.asInstanceOf[Eq[Any]].eqv($e1, $e2) }
            '{ eqProduct((x: T, y: T) => ${eqProductBody('x, 'y)}) }
         case '{ $m: Mirror.SumOf[T] { type MirroredElemTypes = elementTypes }} =>
            val elemInstances = summonAll[elementTypes]
            val eqSumBody: (Expr[T], Expr[T]) => Expr[Boolean] = (x, y) =>
               val ordx = '{ $m.ordinal($x) }
               val ordy = '{ $m.ordinal($y) }
               val elements = Expr.ofList(elemInstances)
               '{ $ordx == $ordy && $elements($ordx).asInstanceOf[Eq[Any]].eqv($x,
         '{ eqSum((x: T, y: T) => ${eqSumBody('x, 'y)}) }
  end derived
end Eq
object Macro3:
  extension [T] (inline x: T)
      inline def === (inline y: T) (using eq: Eq[T]): Boolean = eq.eqv(x, y)
  inline given eqGen[T]: Eq[T] = ${ Eq.derived[T] }
```

5 Metaprogramming

5.1 Overview

The following pages introduce the redesign of metaprogramming in Scala. They introduce the following fundamental facilities:

- 1. inline is a new modifier that guarantees that a definition will be inlined at the point of use. The primary motivation behind inline is to reduce the overhead behind function calls and access to values. The expansion will be performed by the Scala compiler during the Typer compiler phase. As opposed to inlining in some other ecosystems, inlining in Scala is not merely a request to the compiler but is a command. The reason is that inlining in Scala can drive other compile-time operations, like inline pattern matching (enabling type-level programming), macros (enabling compile-time, generative, metaprogramming) and runtime code generation (multi-stage programming).
- 2. Macros are built on two well-known fundamental operations: quotation and splicing. Quotation converts program code to data, specifically, a (tree-like) representation of this code. It is expressed as '{...} for expressions and as '[...] for types. Splicing, expressed as \${...}, goes the other way: it converts a program's representation to program code. Together with inline, these two abstractions allow to construct program code programmatically.
- 3. Staging Where macros construct code at *compile-time*, staging lets programs construct new code at *runtime*. That way, code generation can depend not only on static data but also on data available at runtime. This splits the evaluation of the program in two or more phases or ... stages. Consequently, this method of generative programming is called "Multi-Stage Programming". Staging is built on the same foundations as macros. It uses quotes and splices, but leaves out inline.
- 4. TASTy Reflection Quotations are a "black-box" representation of code. They can be parameterized and composed using splices, but their structure cannot be analyzed from the outside. TASTy reflection gives a way to analyze code structure by partly revealing the representation type of a piece of code in a standard API. The representation type is a form of typed abstract syntax tree, which gives rise to the TASTy moniker.
- 5. TASTy Inspection Typed abstract syntax trees are serialized in a custom compressed binary format stored in .tasty files. TASTy inspection allows to load these files and analyze their content's tree structure.

5.2 Inline \square

5.2.1 Inline Definitions

inline is a new soft modifier that guarantees that a definition will be inlined at the point of use. Example:

```
object Config:
  inline val logging = false

object Logger:

  private var indent = 0

  inline def log[T](msg: String, indentMargin: =>Int)(op: => T): T =
    if Config.logging then
        println(s"${" " * indent}start $msg")
        indent += indentMargin
        val result = op
        indent -= indentMargin
        println(s"${" " * indent}$msg = $result")
        result
        else op
end Logger
```

The Config object contains a definition of the inline value logging. This means that logging is treated as a *constant value*, equivalent to its right-hand side false. The right-hand side of such an inline val must itself be a constant expression. Used in this way, inline is equivalent to Java and Scala 2's final. Note that final, meaning *inlined constant*, is still supported in Scala 3, but will be phased out.

The Logger object contains a definition of the **inline method** log. This method will always be inlined at the point of call.

In the inlined code, an if-then-else with a constant condition will be rewritten to its then- or else-part. Consequently, in the log method above the if Config.logging with Config.logging == true will get rewritten into its then-part.

Here's an example:

```
var indentSetting = 2

def factorial(n: BigInt): BigInt =
   log(s"factorial($n)", indentSetting) {
      if n == 0 then 1
       else n * factorial(n - 1)
   }

If Config.logging == false, this will be rewritten (simplified) to:

def factorial(n: BigInt): BigInt =
   if n == 0 then 1
   else n * factorial(n - 1)
```

As you notice, since neither msg or indentMargin were used, they do not appear

in the generated code for factorial. Also note the body of our log method: the else- part reduces to just an op. In the generated code we do not generate any closures because we only refer to a by-name parameter *once*. Consequently, the code was inlined directly and the call was beta-reduced.

In the true case the code will be rewritten to:

```
def factorial(n: BigInt): BigInt =
  val msg = s"factorial($n)"
  println(s"${" " * indent}start $msg")
  Logger.inline$indent_=(indent.+(indentSetting))
  val result =
     if n == 0 then 1
     else n * factorial(n - 1)
  Logger.inline$indent_=(indent.-(indentSetting))
  println(s"${" " * indent}$msg = $result")
  result
```

Note that the by-value parameter msg is evaluated only once, per the usual Scala semantics, by binding the value and reusing the msg through the body of factorial. Also, note the special handling of the assignment to the private var indent. It is achieved by generating a setter method def inline\$indent_= and calling it instead.

5.2.1.1 Recursive Inline Methods Inline methods can be recursive. For instance, when called with a constant exponent n, the following method for power will be implemented by straight inline code without any loop or recursion.

```
inline def power(x: Double, n: Int): Double =
   if n == 0 then 1.0
   else if n == 1 then x
   else
      val y = power(x, n / 2)
      if n \% 2 == 0 then y * y else y * y * x
power(expr, 10)
// translates to
//
     val x = expr
      val y1 = x * x // ^2
     val \ y2 = y1 * y1 // ^4
//
      val\ y3 = y2 * x // ^5
      y3 * y3
                       // ~10
```

Parameters of inline methods can have an inline modifier as well. This means that actual arguments to these parameters will be inlined in the body of the inline def. inline parameters have call semantics equivalent to by-name parameters but allow for duplication of the code in the argument. It is usually useful when constant values need to be propagated to allow further optimizations/reductions.

The following example shows the difference in translation between by-value, by-name and inline parameters:

```
inline def funkyAssertEquals(actual: Double, expected: =>Double, inline delta: Double
   if (actual - expected).abs > delta then
        throw new AssertionError(s"difference between ${expected} and ${actual} was ]

funkyAssertEquals(computeActual(), computeExpected(), computeDelta())

// translates to

//

// val actual = computeActual()

// def expected = computeExpected()

// if (actual - expected).abs > computeDelta() then

// throw new AssertionError(s"difference between ${expected} and ${actual} was ]
```

5.2.1.2 Rules for Overriding Inline methods can override other non-inline methods. The rules are as follows:

1. If an inline method f implements or overrides another, non-inline method, the inline method can also be invoked at runtime. For instance, consider the scenario:

```
abstract class A:
    def f: Int
    def g: Int = f

class B extends A:
    inline def f = 22
    override inline def g = f + 11

val b = new B

val a: A = b

// inlined invocators

assert(b.f == 22)
assert(b.g == 33)

// dynamic invocations
assert(a.f == 22)
assert(a.g == 33)
```

The inlined invocations and the dynamically dispatched invocations give the same results.

- 2. Inline methods are effectively final.
- 3. Inline methods can also be abstract. An abstract inline method can be implemented only by other inline methods. It cannot be invoked directly:

```
abstract class A: inline def f: Int
```

```
object B extends A:
   inline def f: Int = 22

B.f     // OK
val a: A = B
a.f     // error: cannot inline f() in A.
```

5.2.1.3 Relationship to @inline Scala 2 also defines a @inline annotation which is used as a hint for the backend to inline code. The inline modifier is a more powerful option: Expansion is guaranteed instead of best effort, it happens in the frontend instead of in the backend, and it also applies to recursive methods.

To cross compile between both Scala 3 and Scala 2, we introduce a new <code>@forceInline</code> annotation which is equivalent to the new inline modifier. Note that Scala 2 ignores the <code>@forceInline</code> annotation, so one must use both annotations to guarantee inlining for Scala 3 and at the same time hint inlining for Scala 2 (i.e. <code>@forceInline</code> <code>@inline</code>).

5.2.1.3.1 The definition of constant expression Right-hand sides of inline values and of arguments for inline parameters must be constant expressions in the sense defined by the SLS §6.24, including *platform-specific* extensions such as constant folding of pure numeric computations.

An inline value must have a literal type such as 1 or true.

```
inline val four = 4
// equivalent to
inline val four: 4 = 4
```

It is also possible to have inline vals of types that do not have a syntax, such as Short(4).

```
trait InlineConstants:
   inline val myShort: Short

object Constants extends InlineConstants:
   inline val myShort/*: Short(4)*/ = 4
```

5.2.2 Transparent Inline Methods

Inline methods can additionally be declared **transparent**. This means that the return type of the inline method can be specialized to a more precise type upon expansion. Example:

```
class A
class B extends A:
    def m = true

transparent inline def choose(b: Boolean): A =
```

if b then new A else new B

```
val obj1 = choose(true) // static type is A
val obj2 = choose(false) // static type is B

// obj1.m // compile-time error: `m` is not defined on `A`
obj2.m // OK
```

Here, the inline method choose returns an instance of either of the two types A or B. If choose had not been declared to be transparent, the result of its expansion would always be of type A, even though the computed value might be of the subtype B. The inline method is a "blackbox" in the sense that details of its implementation do not leak out. But if a transparent modifier is given, the expansion is the type of the expanded body. If the argument b is true, that type is A, otherwise it is B. Consequently, calling m on obj2 type-checks since obj2 has the same type as the expansion of choose(false), which is B. Transparent inline methods are "whitebox" in the sense that the type of an application of such a method can be more specialized than its declared return type, depending on how the method expands.

In the following example, we see how the return type of zero is specialized to the singleton type 0 permitting the addition to be ascribed with the correct type 1.

```
transparent inline def zero: Int = 0
val one: 1 = zero + 1
```

5.2.3 Inline Conditionals

An if-then-else expression whose condition is a constant expression can be simplified to the selected branch. Prefixing an if-then-else expression with inline enforces that the condition has to be a constant expression, and thus guarantees that the conditional will always simplify.

Example:

```
inline def update(delta: Int) =
  inline if delta >= 0 then increaseBy(delta)
  else decreaseBy(-delta)
```

A call update(22) would rewrite to increaseBy(22). But if update was called with a value that was not a compile-time constant, we would get a compile time error like the one below:

```
| inline if delta >= 0 then ????
| ^
| cannot reduce inline if
| its condition
| delta >= 0
| is not a constant value
| This location is in code that was inlined at ...
```

5.2.4 Inline Matches

A match expression in the body of an inline method definition may be prefixed by the inline modifier. If there is enough static information to unambiguously take a branch, the expression is reduced to that branch and the type of the result is taken. If not, a compile-time error is raised that reports that the match cannot be reduced.

The example below defines an inline method with a single inline match expression that picks a case based on its static type:

```
transparent inline def g(x: Any): Any =
  inline x match
    case x: String => (x, x) // Tuple2[String, String](x, x)
    case x: Double => x

g(1.0d) // Has type 1.0d which is a subtype of Double
g("test") // Has type (String, String)
```

The scrutinee **x** is examined statically and the inline match is reduced accordingly returning the corresponding value (with the type specialized because **g** is declared **transparent**). This example performs a simple type test over the scrutinee. The type can have a richer structure like the simple ADT below. **toInt** matches the structure of a number in Church-encoding and *computes* the corresponding integer.

```
trait Nat
case object Zero extends Nat
case class Succ[N <: Nat](n: N) extends Nat

transparent inline def toInt(n: Nat): Int =
   inline n match
      case Zero => 0
      case Succ(n1) => toInt(n1) + 1

inline val natTwo = toInt(Succ(Succ(Zero)))
val intTwo: 2 = natTwo
natTwo is inferred to have the singleton type 2.
```

5.2.5 The scala.compiletime Package

The scala.compiletime package contains helper definitions that provide support for compile time operations over values. They are described in the following.

5.2.5.1 constValue, constValueOpt, and the S combinator constValue is a function that produces the constant value represented by a type.

```
import scala.compiletime.{constValue, S}
transparent inline def toIntC[N]: Int =
  inline constValue[N] match
```

```
inline val ctwo = toIntC[2]
```

constValueOpt is the same as constValue, however returning an Option[T] enabling us to handle situations where a value is not present. Note that S is the type of the successor of some singleton type. For example the type S[1] is the singleton type 2.

5.2.5.2 erasedValue So far we have seen inline methods that take terms (tuples and integers) as parameters. What if we want to base case distinctions on types instead? For instance, one would like to be able to write a function defaultValue, that, given a type T, returns optionally the default value of T, if it exists. We can already express this using rewrite match expressions and a simple helper function, scala.compiletime.erasedValue, which is defined as follows:

```
erased def erasedValue[T]: T = ???
```

The erasedValue function *pretends* to return a value of its type argument T. In fact, it would always raise a NotImplementedError exception when called. But the function can in fact never be called, since it is declared erased, so can only be used at compile-time during type checking.

Using erasedValue, we can then define defaultValue as follows:

import scala.compiletime.erasedValue

```
inline def defaultValue[T] =
  inline erasedValue[T] match
     case _: Byte
                     => Some(0: Byte)
     case _: Char
                      => Some(0: Char)
     case : Short
                      => Some(0: Short)
     case _: Int
                      => Some(0)
     case : Long
                      => Some(OL)
     case : Float
                      \Rightarrow Some(0.0f)
     case _: Double => Some(0.0d)
     case _: Boolean => Some(false)
      case _: Unit
                     => Some(())
     case _
                      => None
```

Then:

```
val dInt: Some[Int] = defaultValue[Int]
val dDouble: Some[Double] = defaultValue[Double]
val dBoolean: Some[Boolean] = defaultValue[Boolean]
val dAny: None.type = defaultValue[Any]
```

As another example, consider the type-level version of toInt below: given a type representing a Peano number, return the integer value corresponding to it. Consider

the definitions of numbers as in the *Inline Match* section above. Here is how toIntT can be defined:

```
transparent inline def toIntT[N <: Nat]: Int =
  inline scala.compiletime.erasedValue[N] match
  case _: Zero.type => 0
  case _: Succ[n] => toIntT[n] + 1

inline val two = toIntT[Succ[Succ[Zero.type]]]
```

erasedValue is an erased method so it cannot be used and has no runtime behavior. Since toIntT performs static checks over the static type of N we can safely use it to scrutinize its return type (S[S[Z]] in this case).

5.2.5.3 error The **error** method is used to produce user-defined compile errors during inline expansion. It has the following signature:

```
inline def error(inline msg: String): Nothing
```

If an inline expansion results in a call error(msgStr) the compiler produces an error message containing the given msgStr.

```
import scala.compiletime.{error, code}
inline def fail() =
    error("failed for a reason")

fail() // error: failed for a reason

or
inline def fail(p1: => Any) =
    error(code"failed on: $p1")

fail(identity("foo")) // error: failed on: identity("foo")
```

5.2.5.4 The scala.compiletime.ops package The scala.compiletime.ops package contains types that provide support for primitive operations on singleton types. For example, scala.compiletime.ops.int.* provides support for multiplying two singleton Int types, and scala.compiletime.ops.boolean.&& for the conjunction of two Boolean types. When all arguments to a type in scala.compiletime.ops are singleton types, the compiler can evaluate the result of the operation.

```
import scala.compiletime.ops.int._
import scala.compiletime.ops.boolean._
val conjunction: true && true = true
val multiplication: 3 * 5 = 15
```

Many of these singleton operation types are meant to be used infix (as in SLS §3.2.10).

Since type aliases have the same precedence rules as their term-level equivalents, the operations compose with the expected precedence rules:

```
import scala.compiletime.ops.int._
val x: 1 + 2 * 3 = 7
```

The operation types are located in packages named after the type of the left-hand side parameter: for instance, scala.compiletime.ops.int.+represents addition of two numbers, while scala.compiletime.ops.string.+ represents string concatenation. To use both and distinguish the two types from each other, a match type can dispatch to the correct implementation:

```
import scala.compiletime.ops._
import scala.annotation.infix

type +[X <: Int | String, Y <: Int | String] = (X, Y) match
   case (Int, Int) => int.+[X, Y]
   case (String, String) => string.+[X, Y]

val concat: "a" + "b" = "ab"
val addition: 1 + 1 = 2
```

5.2.6 Summoning Implicits Selectively

It is foreseen that many areas of typelevel programming can be done with rewrite methods instead of implicits. But sometimes implicits are unavoidable. The problem so far was that the Prolog-like programming style of implicit search becomes viral: Once some construct depends on implicit search it has to be written as a logic program itself. Consider for instance the problem of creating a TreeSet[T] or a HashSet[T] depending on whether T has an Ordering or not. We can create a set of implicit definitions like this:

```
trait SetFor[T, S <: Set[T]]

class LowPriority:
   implicit def hashSetFor[T]: SetFor[T, HashSet[T]] = ...

object SetsFor extends LowPriority:
   implicit def treeSetFor[T: Ordering]: SetFor[T, TreeSet[T]] = ...</pre>
```

Clearly, this is not pretty. Besides all the usual indirection of implicit search, we face the problem of rule prioritization where we have to ensure that treeSetFor takes priority over hashSetFor if the element type has an ordering. This is solved (clumsily) by putting hashSetFor in a superclass LowPriority of the object SetsFor where treeSetFor is defined. Maybe the boilerplate would still be acceptable if the crufty code could be contained. However, this is not the case. Every user of the

abstraction has to be parameterized itself with a SetFor implicit. Considering the simple task "I want a TreeSet[T] if T has an ordering and a HashSet[T] otherwise", this seems like a lot of ceremony.

There are some proposals to improve the situation in specific areas, for instance by allowing more elaborate schemes to specify priorities. But they all keep the viral nature of implicit search programs based on logic programming.

By contrast, the new summonFrom construct makes implicit search available in a functional context. To solve the problem of creating the right set, one would use it as follows:

import scala.compiletime.summonFrom

```
inline def setFor[T]: Set[T] = summonFrom {
   case ord: Ordering[T] => new TreeSet[T](using ord)
   case _ => new HashSet[T]
}
```

A summonFrom call takes a pattern matching closure as argument. All patterns in the closure are type ascriptions of the form identifier: Type.

Patterns are tried in sequence. The first case with a pattern x: T such that an implicit value of type T can be summoned is chosen.

Alternatively, one can also use a pattern-bound given instance, which avoids the explicit using clause. For instance, **setFor** could also be formulated as follows:

import scala.compiletime.summonFrom

```
inline def setFor[T]: Set[T] = summonFrom {
   case given Ordering[T] => new TreeSet[T]
   case _ => new HashSet[T]
}
```

summonFrom applications must be reduced at compile time.

Consequently, if we summon an Ordering[String] the code above will return a new instance of TreeSet[String].

```
summon[Ordering[String]]
```

```
println(setFor[String].getClass) // prints class scala.collection.immutable.TreeSe
```

Note summonFrom applications can raise ambiguity errors. Consider the following code with two givens in scope of type A. The pattern match in f will raise an ambiguity error of f is applied.

```
class A
given a1: A = new A
given a2: A = new A
```

```
inline def f: Any = summonFrom {
   case given _: A => ??? // error: ambiguous givens
}
```

5.2.7 summonInline

The shorthand summonInline provides a simple way to write a summon that is delayed until the call is inlined.

```
transparent inline def summonInline[T]: T = summonFrom {
   case t: T => t
}
```

5.2.7.1 Reference For more information, see PR #4768, which explains how summonFrom's predecessor (implicit matches) can be used for typelevel programming and code specialization and PR #7201 which explains the new summonFrom syntax.

5.3 Macros

5.3.1 Macros: Quotes and Splices

Macros are built on two well-known fundamental operations: quotation and splicing. Quotation is expressed as '{...} for expressions and as '[...] for types. Splicing is expressed as \${...}. Additionally, within a quote or a splice we can quote or splice identifiers directly (i.e. 'e and \$e). Readers may notice the resemblance of the two aforementioned syntactic schemes with the familiar string interpolation syntax.

```
println(s"Hello, $name, here is the result of 1 + 1 = $\{1 + 1\}"\}
```

In string interpolation we *quoted* a string and then we *spliced* into it, two others. The first, name, is a reference to a value of type string, and the second is an arithmetic expression that will be *evaluated* followed by the splicing of its string representation.

Quotes and splices in this section allow us to treat code in a similar way, effectively supporting macros. The entry point for macros is an inline method with a top-level splice. We call it a top-level because it is the only occasion where we encounter a splice outside a quote (consider as a quote the compilation-unit at the call-site). For example, the code below presents an inline method assert which calls at compile-time a method assertImpl with a boolean expression tree as argument. assertImpl evaluates the expression and prints it again in an error message if it evaluates to false.

```
import scala.quoted._
inline def assert(inline expr: Boolean): Unit =
   ${ assertImpl('expr) }

def assertImpl(expr: Expr[Boolean])(using Quotes) = '{
```

If e is an expression, then '{e} represents the typed abstract syntax tree representing e. If T is a type, then '[T] represents the type structure representing T. The precise definitions of "typed abstract syntax tree" or "type-structure" do not matter for now, the terms are used only to give some intuition. Conversely, \${e} evaluates the expression e, which must yield a typed abstract syntax tree or type structure, and embeds the result as an expression (respectively, type) in the enclosing program.

Quotations can have spliced parts in them; in this case the embedded splices are evaluated and embedded as part of the formation of the quotation.

Quotes and splices can also be applied directly to identifiers. An identifier x starting with a that appears inside a quoted expression or type is treated as a splice x. Analogously, an quoted identifier 'x that appears inside a splice is treated as a quote 'x. See the Syntax section below for details.

Quotes and splices are duals of each other. For arbitrary expressions **e** and types **T** we have:

```
${'\{e\}} = e
'\{\$\{e\}} = e
$\{'\[T\]\} = T
'\[\$\{T\}\] = T
```

5.3.2 Types for Quotations

The type signatures of quotes and splices can be described using two fundamental types:

- Expr[T]: abstract syntax trees representing expressions of type T
- Type [T]: type structures representing type T.

Quoting takes expressions of type T to expressions of type Expr[T] and it takes types T to expressions of type Type[T]. Splicing takes expressions of type Expr[T] to expressions of type T and it takes expressions of type Type[T] to types T.

The two types can be defined in package scala.quoted as follows:

```
package scala.quoted
sealed abstract class Expr[+T]
sealed abstract class Type[T]
```

Both Expr and Type are abstract and sealed, so all constructors for these types are provided by the system. One way to construct values of these types is by quoting, the other is by type-specific lifting operations that will be discussed later on.

5.3.3 The Phase Consistency Principle

A fundamental *phase consistency principle* (PCP) regulates accesses to free variables in quoted and spliced code:

• For any free variable reference \mathbf{x} , the number of quoted scopes and the number of spliced scopes between the reference to \mathbf{x} and the definition of \mathbf{x} must be equal.

Here, this-references count as free variables. On the other hand, we assume that all imports are fully expanded and that <code>root</code> is not a free variable. So references to global definitions are allowed everywhere.

The phase consistency principle can be motivated as follows: First, suppose the result of a program P is some quoted text ' $\{\ldots x \ldots \}$ that refers to a free variable x in P. This can be represented only by referring to the original variable x. Hence, the result of the program will need to persist the program state itself as one of its parts. We don't want to do this, hence this situation should be made illegal. Dually, suppose a top-level part of a program is a spliced text $\{\ldots x \ldots \}$ that refers to a free variable x in P. This would mean that we refer during construction of P to a value that is available only during execution of P. This is of course impossible and therefore needs to be ruled out. Now, the small-step evaluation of a program will reduce quotes and splices in equal measure using the cancellation rules above. But it will neither create nor remove quotes or splices individually. So the PCP ensures that program elaboration will lead to neither of the two unwanted situations described above.

In what concerns the range of features it covers, this form of macros introduces a principled metaprogramming framework that is quite close to the MetaML family of languages. One difference is that MetaML does not have an equivalent of the PCP - quoted code in MetaML can access variables in its immediately enclosing environment, with some restrictions and caveats since such accesses involve serialization. However, this does not constitute a fundamental gain in expressiveness.

5.3.4 From Exprs to Functions and Back

It is possible to convert any $Expr[T \Rightarrow R]$ into $Expr[T] \Rightarrow Expr[R]$ and back. These conversions can be implemented as follows:

```
def to[T: Type, R: Type](f: Expr[T] => Expr[R])(using Quotes): Expr[T => R] =
   '{ (x: T) => ${ f('x) } }

def from[T: Type, R: Type](f: Expr[T => R])(using Quotes): Expr[T] => Expr[R] =
   (x: Expr[T]) => '{ $f($x) }
```

Note how the fundamental phase consistency principle works in two different directions here for \mathbf{f} and \mathbf{x} . In the method \mathbf{to} , the reference to \mathbf{f} is legal because it is quoted, then spliced, whereas the reference to \mathbf{x} is legal because it is spliced, then quoted.

They can be used as follows:

```
val f1: Expr[Int => String] =
   to((x: Expr[Int]) => '{ $x.toString }) // '{ (x: Int) => x.toString }

val f2: Expr[Int] => Expr[String] =
   from('{ (x: Int) => x.toString }) // (x: Expr[Int]) => '{ ((x: Int) => x.toString ) } (2) }
```

One limitation of from is that it does not -reduce when a lambda is called immediately, as evidenced in the code { ((x: Int) => x.toString)(2) }. In some cases we want to remove the lambda from the code, for this we provide the method Expr.betaReduce that turns a tree describing a function into a function mapping trees to trees.

```
object Expr:
    ...
    def betaReduce[...](...)(...): ... = ...
```

The definition of Expr.betaReduce(f)(x) is assumed to be functionally the same as '{(\$f)(\$x)}, however it should optimize this call by returning the result of beta-reducing f(x) if f is a known lambda expression. Expr.betaReduce distributes applications of Expr over function arrows:

```
 \texttt{Expr.betaReduce(\_): Expr[(T1, ..., Tn) => R] => ((Expr[T1], ..., Expr[Tn]) => Expr[Tn]) => Expr[Tn]) => Expr[Tn] => Expr[
```

5.3.5 Lifting Types

Types are not directly affected by the phase consistency principle. It is possible to use types defined at any level in any other level. But, if a type is used in a subsequent stage it will need to be lifted to a Type. The resulting value of Type will be subject to PCP. Indeed, the definition of to above uses T in the next stage, there is a quote but no splice between the parameter binding of T and its usage. But the code can be rewritten by adding a binding of a Type[T] tag:

```
def to[T, R](f: Expr[T] => Expr[R])(using Type[T], Type[R], Quotes): Expr[T => R] = ^{\prime}{ (x: T) => ${ f('x) }}
```

In this version of to, the type of x is now the result of splicing the Type value t. This operation is splice correct – there is one quote and one splice between the use of t and its definition.

To avoid clutter, the Scala implementation tries to convert any type reference to a type T in subsequent phases to a type-splice, by rewriting T to summon[Type[T]].Underlying. For instance, the user-level definition of to:

```
 def \ to[T, R](f: Expr[T] \Rightarrow Expr[R])(using \ t: Type[T], \ r: Type[R])(using \ Quotes): Expr[R](x: T) \Rightarrow $\{ f('x) \} \}
```

would be rewritten to

```
def to[T, R](f: Expr[T] => Expr[R])(using t: Type[T], r: Type[R])(using Quotes): Ex
   '{ (x: t.Underlying) => ${ f('x) } }
```

The summon query succeeds because there is a given instance of type Type[T] available (namely the given parameter corresponding to the context bound: Type), and the reference to that value is phase-correct. If that was not the case, the phase inconsistency for T would be reported as an error.

5.3.6 Lifting Expressions

Consider the following implementation of a staged interpreter that implements a compiler through staging.

```
import scala.quoted._
enum Exp:
    case Num(n: Int)
    case Plus(e1: Exp, e2: Exp)
    case Var(x: String)
    case Let(x: String, e: Exp, in: Exp)

import Exp._
```

The interpreted language consists of numbers Num, addition Plus, and variables Var which are bound by Let. Here are two sample expressions in the language:

```
val exp = Plus(Plus(Num(2), Var("x")), Num(4))
val letExp = Let("x", Num(3), exp)
```

Here's a compiler that maps an expression given in the interpreted language to quoted Scala code of type Expr[Int]. The compiler takes an environment that maps variable names to Scala Exprs.

```
import scala.quoted._
```

```
def compile(e: Exp, env: Map[String, Expr[Int]])(using Quotes): Expr[Int] =
    e match
    case Num(n) =>
        Expr(n)
    case Plus(e1, e2) =>
        '{ ${ compile(e1, env) } + ${ compile(e2, env) } }
    case Var(x) =>
        env(x)
    case Let(x, e, body) =>
        '{ val y = ${ compile(e, env) }; ${ compile(body, env + (x -> 'y)) } }
```

Running compile(letExp, Map()) would yield the following Scala code:

```
'{ val y = 3; (2 + y) + 4 }
```

The body of the first clause, case Num(n) => Expr(n), looks suspicious. n is declared as an Int, yet it is converted to an Expr[Int] with Expr(). Shouldn't n be quoted? In fact this would not work since replacing n by 'n in the clause would not be phase correct.

The Expr.apply method is defined in package quoted:

```
package quoted

object Expr:
    ...
    def apply[T: ToExpr](x: T)(using Quotes): Expr[T] =
        summon[ToExpr[T]].toExpr(x)
```

This method says that values of types implementing the ToExpr type class can be converted to Expr values using Expr.apply.

Scala 3 comes with given instances of ToExpr for several types including Boolean, String, and all primitive number types. For example, Int values can be converted to Expr[Int] values by wrapping the value in a Literal tree node. This makes use of the underlying tree representation in the compiler for efficiency. But the ToExpr instances are nevertheless not *magic* in the sense that they could all be defined in a user program without knowing anything about the representation of Expr trees. For instance, here is a possible instance of ToExpr[Boolean]:

```
given ToExpr[Boolean] with
  def toExpr(b: Boolean) =
    if b then '{ true } else '{ false }
```

Once we can lift bits, we can work our way up. For instance, here is a possible implementation of ToExpr[Int] that does not use the underlying tree machinery:

```
given ToExpr[Int] with
  def toExpr(n: Int) = n match
    case Int.MinValue => '{ Int.MinValue }
    case _ if n < 0 => '{ - ${ toExpr(-n) } }
    case 0 => '{ 0 }
    case _ if n % 2 == 0 => '{ ${ toExpr(n / 2) } * 2 }
    case _ if n % 2 == 0 => '{ ${ toExpr(n / 2) } * 2 }
    case _ if n % 2 == 0 => '{ ${ toExpr(n / 2) } * 2 }
    case _ if n % 2 == 0 => '{ ${ toExpr(n / 2) } * 2 }
```

Since ToExpr is a type class, its instances can be conditional. For example, a List is liftable if its element type is:

```
given [T: ToExpr : Type]: ToExpr[List[T]] with
  def toExpr(xs: List[T]) = xs match
    case head :: tail => '{ ${ Expr(head) } :: ${ toExpr(tail) } }
    case Nil => '{ Nil: List[T] }
```

In the end, ToExpr resembles very much a serialization framework. Like the latter it can be derived systematically for all collections, case classes and enums. Note also that the synthesis of *type-tag* values of type Type[T] is essentially the type-level analogue of lifting.

Using lifting, we can now give the missing definition of **showExpr** in the introductory example:

```
def showExpr[T](expr: Expr[T])(using Quotes): Expr[String] =
  val code: String = expr.show
  Expr(code)
```

That is, the showExpr method converts its Expr argument to a string (code), and lifts the result back to an Expr[String] using Expr.apply.

5.3.7 Lifting Types

The previous section has shown that the metaprogramming framework has to be able to take a type T and convert it to a type tree of type Type[T] that can be reified. This means that all free variables of the type tree refer to types and values defined in the current stage.

For a reference to a global class, this is easy: Just issue the fully qualified name of the class. Members of reifiable types are handled by just reifying the containing type together with the member name. But what to do for references to type parameters or local type definitions that are not defined in the current stage? Here, we cannot construct the Type[T] tree directly, so we need to get it from a recursive implicit search. For instance, to implement

```
summon[Type[List[T]]]
```

where T is not defined in the current stage, we construct the type constructor of List applied to the splice of the result of searching for a given instance for Type [T]:

```
'[ List[ ${ summon[Type[T]] } ] ]
```

This is exactly the algorithm that Scala 2 uses to search for type tags. In fact Scala 2's type tag feature can be understood as a more ad-hoc version of quoted. Type. As was the case for type tags, the implicit search for a quoted. Type is handled by the compiler, using the algorithm sketched above.

5.3.8 Relationship with inline

Seen by itself, principled metaprogramming looks more like a framework for runtime metaprogramming than one for compile-time metaprogramming with macros. But combined with Scala 3's inline feature it can be turned into a compile-time system. The idea is that macro elaboration can be understood as a combination of a macro library and a quoted program. For instance, here's the assert macro again together with a program that calls assert.

object Macros:

```
inline def assert(inline expr: Boolean): Unit =
    ${ assertImpl('expr) }

def assertImpl(expr: Expr[Boolean])(using Quotes) =
    val failMsg: Expr[String] = Expr("failed assertion: " + expr.show)
    '{ if !($expr) then throw new AssertionError($failMsg) }
```

```
@main def program =
  val x = 1
  Macros.assert(x != 0)
```

Inlining the assert function would give the following program:

```
@main def program =
  val x = 1
  ${ Macros.assertImpl('{ x != 0) } }
```

The example is only phase correct because Macros is a global value and as such not subject to phase consistency checking. Conceptually that's a bit unsatisfactory. If the PCP is so fundamental, it should be applicable without the global value exception. But in the example as given this does not hold since both assert and program call assertImpl with a splice but no quote.

However, one could argue that the example is really missing an important aspect: The macro library has to be compiled in a phase prior to the program using it, but in the code above, macro and program are defined together. A more accurate view of macros would be to have the user program be in a phase after the macro definitions, reflecting the fact that macros have to be defined and compiled before they are used. Hence, conceptually the program part should be treated by the compiler as if it was quoted:

```
@main def program = '{
   val x = 1
   ${ Macros.assertImpl('{ x != 0 }) }
}
```

If program is treated as a quoted expression, the call to Macro.assertImpl becomes phase correct even if macro library and program are conceptualized as local definitions.

But what about the call from assert to assertImpl? Here, we need a tweak of the typing rules. An inline function such as assert that contains a splice operation outside an enclosing quote is called a *macro*. Macros are supposed to be expanded in a subsequent phase, i.e. in a quoted context. Therefore, they are also type checked as if they were in a quoted context. For instance, the definition of assert is typechecked as if it appeared inside quotes. This makes the call from assert to assertImpl phase-correct, even if we assume that both definitions are local.

The inline modifier is used to declare a val that is either a constant or is a parameter that will be a constant when instantiated. This aspect is also important for macro expansion.

To get values out of expressions containing constants Expr provides the method value (or valueOrError). This will convert the Expr[T] into a Some[T] (or T) when the expression contains value. Otherwise it will retrun None (or emit an error). To avoid having incidental val bindings generated by the inlining of the

def it is recommended to use an inline parameter. To illustrate this, consider an implementation of the power function that makes use of a statically known exponent:

```
inline def power(x: Double, inline n: Int) = ${ powerCode('x, 'n) }

private def powerCode(x: Expr[Double], n: Expr[Int])(using Quotes): Expr[Double] =
    n.value match
    case Some(m) => powerCode(x, m)
    case None => '{ Math.pow($x, $n.toDouble) }

private def powerCode(x: Expr[Double], n: Int)(using Quotes): Expr[Double] =
    if n == 0 then '{ 1.0 }
    else if n == 1 then x
    else if n % 2 == 0 then '{ val y = $x * $x; ${ powerCode('y, n / 2) } }
    else '{ $x * ${ powerCode(x, n - 1) } }
}
```

5.3.9 Scope Extrusion

Quotes and splices are duals as far as the PCP is concerned. But there is an additional restriction that needs to be imposed on splices to guarantee soundness: code in splices must be free of side effects. The restriction prevents code like this:

```
var x: Expr[T] = ...
'{ (y: T) => ${ x = 'y; 1 } }
```

This code, if it was accepted, would extrude a reference to a quoted variable y from its scope. This would subsequently allow access to a variable outside the scope where it is defined, which is likely problematic. The code is clearly phase consistent, so we cannot use PCP to rule it out. Instead we postulate a future effect system that can guarantee that splices are pure. In the absence of such a system we simply demand that spliced expressions are pure by convention, and allow for undefined compiler behavior if they are not. This is analogous to the status of pattern guards in Scala, which are also required, but not verified, to be pure.

Multi-Stage Programming introduces one additional method where you can expand code at runtime with a method run. There is also a problem with that invokation of run in splices. Consider the following expression:

```
'{ (x: Int) => ${ run('x); 1 } }
```

This is again phase correct, but will lead us into trouble. Indeed, evaluating the splice will reduce the expression run('x) to x. But then the result

```
'{ (x: Int) => ${ x; 1 } }
```

is no longer phase correct. To prevent this soundness hole it seems easiest to classify run as a side-effecting operation. It would thus be prevented from appearing in splices. In a base language with side effects we would have to do this anyway: Since run runs arbitrary code it can always produce a side effect if the code it runs produces one.

5.3.10 Example Expansion

Assume we have two methods, one map that takes an Expr[Array[T]] and a function f and one sum that performs a sum by delegating to map.

object Macros:

```
def map[T](arr: Expr[Array[T]], f: Expr[T] => Expr[Unit])
              (using Type[T], Quotes): Expr[Unit] = '{
      var i: Int = 0
      while i < ($arr).length do
         val element: T = (\$arr)(i)
         ${f('element)}
         i += 1
   }
   def sum(arr: Expr[Array[Int]])(using Quotes): Expr[Int] = '{
      var sum = 0
      \{ map(arr, x => '\{sum += $x\}) \}
      sum
   }
   inline def sum m(arr: Array[Int]): Int = ${sum('arr)}
end Macros
A call to sum_m(Array(1,2,3)) will first inline sum_m:
val arr: Array[Int] = Array.apply(1, [2,3 : Int]:Int*)
${_root_.Macros.sum('arr)}
then it will splice sum:
val arr: Array[Int] = Array.apply(1, [2,3 : Int]:Int*)
var sum = 0
\{ map(arr, x => '\{sum += $x\}) \}
sum
then it will inline map:
val arr: Array[Int] = Array.apply(1, [2,3 : Int]:Int*)
var sum = 0
val f = x => '{sum += $x}
${ _root_.Macros.map(arr, 'f)('[Int])}
sum
then it will expand and splice inside quotes map:
val arr: Array[Int] = Array.apply(1, [2,3 : Int]:Int*)
```

```
var sum = 0
val f = x => '{sum += $x}
var i: Int = 0
while i < arr.length do
   val element: Int = (arr)(i)
   sum += element
   i += 1
sum
Finally cleanups and dead code elimination:
val arr: Array[Int] = Array.apply(1, [2,3 : Int]:Int*)
var sum = 0
var i: Int = 0
while i < arr.length do
   val element: Int = arr(i)
   sum += element
   i += 1
sum
```

5.3.11 Find implicits within a macro

Similarly to the summonFrom construct, it is possible to make implicit search available in a quote context. For this we simply provide scala.quoted.Expr.summon:

```
import scala.collection.immutable.{ TreeSet, HashSet }
inline def setFor[T]: Set[T] = ${ setForExpr[T] }

def setForExpr[T: Type](using Quotes): Expr[Set[T]] =
    Expr.summon[Ordering[T]] match
    case Some(ord) => '{ new TreeSet[T]()($ord) }
    case _ => '{ new HashSet[T] }
```

5.3.12 Relationship with Whitebox Inline

Inline documents inlining. The code below introduces a whitebox inline method that can calculate either a value of type Int or a value of type String.

```
transparent inline def defaultOf(inline str: String) =
    ${ defaultOfImpl('str) }

def defaultOfImpl(strExpr: Expr[String])(using Quotes): Expr[Any] =
    strExpr.valueOrError match
        case "int" => '{1}
        case "string" => '{"a"}

// in a separate file
```

```
val a: Int = defaultOf("int")
val b: String = defaultOf("string")
```

5.3.13 Defining a macro and using it in a single project

It is possible to define macros and use them in the same project as long as the implementation of the macros does not have run-time dependencies on code in the file where it is used. It might still have compile-time dependencies on types and quoted code that refers to the use-site file.

To provide this functionality Scala 3 provides a transparent compilation mode where files that try to expand a macro but fail because the macro has not been compiled yet are suspended. If there are any suspended files when the compilation ends, the compiler will automatically restart compilation of the suspended files using the output of the previous (partial) compilation as macro classpath. In case all files are suspended due to cyclic dependencies the compilation will fail with an error.

5.3.14 Pattern matching on quoted expressions

It is possible to deconstruct or extract values out of Expr using pattern matching. scala.quoted contains objects that can help extracting values from Expr.

- scala.quoted.Expr/scala.quoted.Exprs: matches an expression of a value (or list of values) and returns the value (or list of values).
- scala.quoted.Const/scala.quoted.Consts: Same as Expr/Exprs but only works on primitive values.
- scala.quoted.Varargs: matches an explicit sequence of expressions and returns them. These sequences are useful to get individual Expr[T] out of a varargs expression of type Expr[Seq[T]].

These could be used in the following way to optimize any call to sum that has statically known values.

```
inline def sum(inline args: Int*): Int = ${ sumExpr('args) }
private def sumExpr(argsExpr: Expr[Seq[Int]])(using Quotes): Expr[Int] =
    argsExpr match
    case Varargs(args @ Exprs(argValues)) =>
        // args is of type Seq[Expr[Int]]
        // argValues is of type Seq[Int]
        Expr(argValues.sum) // precompute result of sum
    case Varargs(argExprs) => // argExprs is of type Seq[Expr[Int]]
        val staticSum: Int = argExprs.map(_.value.getOrElse(0)).sum
        val dynamicSum: Seq[Expr[Int]] = argExprs.filter(_.value.isEmpty)
        dynamicSum.foldLeft(Expr(staticSum))((acc, arg) => '{ $acc + $arg })
    case _ =>
        '{ $argsExpr.sum }
```

5.3.14.1 Quoted patterns Quoted patterns allow deconstructing complex code that contains a precise structure, types or methods. Patterns '{ ...} can be placed in any location where Scala expects a pattern.

```
For example
optimize {
   sum(sum(1, a, 2), 3, b)
} // should be optimized to 6 + a + b
def sum(args: Int*): Int = args.sum
inline def optimize(inline arg: Int): Int = ${ optimizeExpr('arg) }
private def optimizeExpr(body: Expr[Int])(using Quotes): Expr[Int] =
   body match
      // Match a call to sum without any arguments
      case '{ sum() } => Expr(0)
      // Match a call to sum with an argument $n of type Int.
      // n will be the Expr[Int] representing the argument.
      case '{ sum(\$n) } => n
      // Match a call to sum and extracts all its args in an `Expr[Seq[Int]]`
      case '{ sum(${Varargs(args)}: *) } => sumExpr(args)
      case body => body
private def sumExpr(args1: Seq[Expr[Int]])(using Quotes): Expr[Int] =
   def flatSumArgs(arg: Expr[Int]): Seq[Expr[Int]] = arg match
      case '{ sum(${Varargs(subArgs)}: _*) } => subArgs.flatMap(flatSumArgs)
      case arg => Seq(arg)
   val args2 = args1.flatMap(flatSumArgs)
   val staticSum: Int = args2.map( .value.getOrElse(0)).sum
   val dynamicSum: Seq[Expr[Int]] = args2.filter(_.value.isEmpty)
   dynamicSum.foldLeft(Expr(staticSum))((acc, arg) => '{ $acc + $arg })
```

5.3.14.2 Recovering precise types using patterns Sometimes it is necessary to get a more precise type for an expression. This can be achived using the following pattern match.

```
def f(expr: Expr[Any])(using Quotes) = expr match
  case '{ $x: t } =>
    // If the pattern match succeeds, then there is
    // some type `t` such that
    // - `x` is bound to a variable of type `Expr[t]`
    // - `t` is bound to a new type `t` and a given
    // instance `Type[t]` is provided for it
    // That is, we have `x: Expr[t]` and `given Type[t]`,
    // for some (unknown) type `t`.
```

This might be used to then perform an implicit search as in:

```
extension (inline sc: StringContext)
```

```
inline def showMe(inline args: Any*): String = ${ showMeExpr('sc, 'args) }
private def showMeExpr(sc: Expr[StringContext], argsExpr: Expr[Seq[Any]])(using Queen
   argsExpr match
      case Varargs(argExprs) =>
         val argShowedExprs = argExprs.map {
            case '{ $arg: tp } =>
               val showTp = Type.of[Show[tp]]
               Expr.summon(using showTp) match
                  case Some(showExpr) =>
                      '{ $showExpr.show($arg) }
                  case None =>
                     report.error(s"could not find implicit for ${Type.show[Show[tp
         }
         val newArgsExpr = Varargs(argShowedExprs)
         '{ $sc.s($newArgsExpr: _*) }
      case =>
         // `new StringContext(\dots).showMeExpr(args: \_*) ` not an explicit `showMeExpr(args: \_*)
         report.error(s"Args must be explicit", argsExpr)
trait Show[-T]:
   def show(x: T): String
// in a different file
given Show[Boolean] with
   def show(b: Boolean) = "boolean!"
println(showMe"${true}")
```

5.3.14.3 Open code patterns Quoted pattern matching also provides higher-order patterns to match open terms. If a quoted term contains a definition, then the rest of the quote can refer to this definition.

```
'{
    val x: Int = 4
    x * x
}
```

To match such a term we need to match the definition and the rest of the code, but we need to explicitly state that the rest of the code may refer to this definition.

```
case '{ val y: Int = $x; $body(y): Int } =>
```

Here \$x will match any closed expression while \$body(y) will match an expression that is closed under y. Then the subexpression of type Expr[Int] is bound to body as an Expr[Int => Int]. The extra argument represents the references to y. Usually this expression is used in combination with Expr.betaReduce to replace

the extra argument.

We can also close over several bindings using \$b(a1, a2, ..., an). To match an actual application we can use braces on the function part \${b}(a1, a2, ..., an).

5.3.15 More details

More details

5.4 Multi-Stage Programming

The framework expresses at the same time compile-time metaprogramming and multi-stage programming. We can think of compile-time metaprogramming as a two stage compilation process: one that we write the code in top-level splices, that will be used for code generation (macros) and one that will perform all necessary evaluations at compile-time and an object program that we will run as usual. What if we could synthesize code at run-time and offer one extra stage to the programmer? Then we can have a value of type Expr[T] at run-time that we can essentially treat as a typed-syntax tree that we can either *show* as a string (pretty-print) or compile and run. If the number of quotes exceeds the number of splices by more than one (effectively handling at run-time values of type Expr[Expr[T]], Expr[Expr[T]]], ...) then we talk about Multi-Stage Programming.

The motivation behind this *paradigm* is to let runtime information affect or guide code-generation.

Intuition: The phase in which code is run is determined by the difference between the number of splice scopes and quote scopes in which it is embedded.

• If there are more splices than quotes, the code is run at compile-time i.e. as a macro. In the general case, this means running an interpreter that evaluates the code, which is represented as a typed abstract syntax tree. The interpreter

can fall back to reflective calls when evaluating an application of a previously compiled method. If the splice excess is more than one, it would mean that a macro's implementation code (as opposed to the code it expands to) invokes other macros. If macros are realized by interpretation, this would lead to towers of interpreters, where the first interpreter would itself interpret an interpreter code that possibly interprets another interpreter and so on.

- If the number of splices equals the number of quotes, the code is compiled and run as usual.
- If the number of quotes exceeds the number of splices, the code is staged. That is, it produces a typed abstract syntax tree or type structure at run-time. A quote excess of more than one corresponds to multi-staged programming.

Providing an interpreter for the full language is quite difficult, and it is even more difficult to make that interpreter run efficiently. So we currently impose the following restrictions on the use of splices.

- 1. A top-level splice must appear in an inline method (turning that method into a macro)
- 2. The splice must call a previously compiled method passing quoted arguments, constant arguments or inline arguments.
- 3. Splices inside splices (but no intervening quotes) are not allowed.

5.4.1 API

The framework as discussed so far allows code to be staged, i.e. be prepared to be executed at a later stage. To run that code, there is another method in class Expr called run. Note that \$ and run both map from Expr[T] to T but only \$ is subject to the PCP, whereas run is just a normal method. run provides a Quotes that can be used to show the expression in its scope. On the other hand withQuotes provides a Quotes without evaluating the expression.

```
package scala.quoted.staging
```

```
def run[T](expr: Quotes ?=> Expr[T])(using toolbox: Toolbox): T = ...
def withQuotes[T](thunk: Quotes ?=> T)(using toolbox: Toolbox): T = ...
```

5.4.2 Create a new Scala 3 project with staging enabled

```
sbt new scala/scala3-staging.g8
```

From scala/scala3-staging.g8.

It will create a project with the necessary dependencies and some examples.

In case you prefer to create the project on your own, make sure to define the following dependency in your build.sbt build definition

libraryDependencies += "org.scala-lang" %% "scala3-staging" % scalaVersion.value and in case you use scalac/scala directly, then use the -with-compiler flag for both:

```
scalac -with-compiler -d out Test.scala
scala -with-compiler -classpath out Test
```

5.4.3 Example

Now take exactly the same example as in Macros. Assume that we do not want to pass an array statically but generate code at run-time and pass the value, also at run-time. Note, how we make a future-stage function of type Expr[Array[Int] => Int] in line 6 below. Using run { . . . } we can evaluate an expression at runtime. Within the scope of run we can also invoke show on an expression to get a source-like representation of the expression.

```
import scala.quoted.staging._

// make available the necessary toolbox for runtime code generation
given Toolbox = Toolbox.make(getClass.getClassLoader)

val f: Array[Int] => Int = run {
   val stagedSum: Expr[Array[Int] => Int] =
        '{ (arr: Array[Int]) => ${sum('arr)}}
   println(stagedSum.show) // Prints "(arr: Array[Int]) => { var sum = 0; ... }"
   stagedSum
}

f.apply(Array(1, 2, 3)) // Returns 6
```

5.5 TASTy Reflect

TASTy Reflect enables inspection and construction of Typed Abstract Syntax Trees (Typed-AST). It may be used on quoted expressions (quoted.Expr) and quoted types (quoted.Type) from Macros or on full TASTy files.

If you are writing macros, please first read Macros. You may find all you need without using TASTy Reflect.

5.5.1 API: From quotes and splices to TASTy reflect trees and back

With quoted.Expr and quoted.Type we can compute code but also analyze code by inspecting the ASTs. Macros provide the guarantee that the generation of code will be type-correct. Using TASTy Reflect will break these guarantees and may fail at macro expansion time, hence additional explicit checks must be done.

To provide reflection capabilities in macros we need to add an implicit parameter of type scala.quoted.Quotes and import quotes.reflect._ from it in the scope

where it is used.

```
import scala.quoted._
inline def natConst(inline x: Int): Int = ${natConstImpl('{x})}
def natConstImpl(x: Expr[Int])(using Quotes): Expr[Int] =
  import quotes.reflect._
```

5.5.1.1 Extractors import quotes.reflect. will provide all extractors and methods on TASTy Reflect trees. For example the Literal() extractor used below.

```
def natConstImpl(x: Expr[Int])(using Quotes): Expr[Int] =
  import quotes.reflect._
  val xTree: Term = x.asTerm
  xTree match
    case Inlined(_, _, Literal(IntConstant(n))) =>
      if n <= 0 then
        report.error("Parameter must be natural number")
        '{0}
    else
        xTree.asExprOf[Int]
    case _ =>
      report.error("Parameter must be a known constant")
    '{0}
```

We can easily know which extractors are needed using Printer.TreeStructure.show, which returns the string representation the structure of the tree. Other printers can also be found in the Printer module.

```
xTree.show(using Printer.TreeStructure)
// or
Printer.TreeStructure.show(xTree)
```

The methods quotes.reflect.Term.{asExpr, asExprOf} provide a way to go back to a quoted.Expr. Note that asExpr returns a Expr[Any]. On the other hand asExprOf[T] returns a Expr[T], if the type does not conform to it an exception will be thrown at runtime.

5.5.1.2 Positions The ast in the context provides a rootPosition value. It corresponds to the expansion site for macros. The macro authors can obtain various information about that expansion site. The example below shows how we can obtain position information such as the start line, the end line or even the source code at the expansion point.

```
def macroImpl()(quotes: Quotes): Expr[Unit] =
  import quotes.reflect._
  val pos = rootPosition
```

```
val path = pos.sourceFile.jpath.toString
val start = pos.start
val end = pos.end
val startLine = pos.startLine
val endLine = pos.endLine
val startColumn = pos.startColumn
val endColumn = pos.endColumn
val sourceCode = pos.sourceCode
```

5.5.1.3 Tree Utilities scala.tasty.reflect contains three facilities for tree traversal and transformation.

TreeAccumulator ties the knot of a traversal. By calling foldOver(x, tree)) we can dive into the tree node and start accumulating values of type X (e.g., of type List[Symbol] if we want to collect symbols). The code below, for example, collects the pattern variables of a tree.

```
def collectPatternVariables(tree: Tree)(implicit ctx: Context): List[Symbol] =
  val acc = new TreeAccumulator[List[Symbol]]:
    def apply(syms: List[Symbol], tree: Tree)(implicit ctx: Context) = tree match
        case Bind(_, body) => apply(tree.symbol :: syms, body)
        case _ => foldOver(syms, tree)
    acc(Nil, tree)
```

A TreeTraverser extends a TreeAccumulator and performs the same traversal but without returning any value. Finally a TreeMap performs a transformation.

5.5.1.3.1 Let scala.tasty.Reflection also offers a method let that allows us to bind the rhs (right-hand side) to a val and use it in body. Additionally, lets binds the given terms to names and allows to use them in the body. Their type definitions are shown below:

```
def let(rhs: Term)(body: Ident => Term): Term = ...
def lets(terms: List[Term])(body: List[Term] => Term): Term = ...
```

5.5.2 More Examples

• Start experimenting with TASTy Reflect (link)

5.6 TASTy Inspection

```
libraryDependencies += "org.scala-lang" %% "scala3-tasty-inspector" % scalaVersion.
```

TASTy files contain the full typed tree of a class including source positions and documentation. This is ideal for tools that analyze or extract semantic information from the code. To avoid the hassle of working directly with the TASTy file we

provide the TastyInspector which loads the contents and exposes it through the TASTy reflect API.

5.6.1 Inspecting TASTy files

To inspect the TASTy Reflect trees of a TASTy file a consumer can be defined in the following way.

```
import scala.tasty.Reflection
import scala.tasty.file._

class Consumer extends TastyInspector:
    final def apply(reflect: Reflection)(root: reflect.Tree): Unit =
        import reflect._
        // Do something with the tree
```

Then the consumer can be instantiated with the following code to get the tree of the class foo.Bar for a foo in the classpath.

Note that if we need to run the main (in the example below defined in an object called Test) after compilation we need to make the compiler available to the runtime:

```
scalac -d out Test.scala
scala -with-compiler -classpath out Test
```

5.6.2 Template project

Using sbt version 1.1.5+, do:

```
sbt new scala/scala3-tasty-inspector.g8
```

in the folder where you want to clone the template.

5.7 Erased Terms

5.7.1 Why erased terms?

Let's describe the motivation behind erased terms with an example. In the following we show a simple state machine which can be in a state On or Off. The machine can change state from Off to On with turnedOn only if it is currently Off. This last constraint is captured with the IsOff[S] contextual evidence which only exists for IsOff[Off]. For example, not allowing calling turnedOn on in an On state as we would require an evidence of type IsOff[On] that will not be found.

```
sealed trait State
final class On extends State
final class Off extends State
```

```
@implicitNotFound("State must be Off")
class IsOff[S <: State]
object IsOff:
    given isOff: IsOff[Off] = new IsOff[Off]

class Machine[S <: State]:
    def turnedOn(using IsOff[S]): Machine[On] = new Machine[On]

val m = new Machine[Off]
m.turnedOn
m.turnedOn.turnedOn // ERROR
///
// State must be Off</pre>
```

Note that in the code above the actual context arguments for IsOff are never used at runtime; they serve only to establish the right constraints at compile time. As these terms are never used at runtime there is not real need to have them around, but they still need to be present in some form in the generated code to be able to do separate compilation and retain binary compatibility. We introduce *erased terms* to overcome this limitation: we are able to enforce the right constrains on terms at compile time. These terms have no run time semantics and they are completely erased.

5.7.2 How to define erased terms?

Parameters of methods and functions can be declared as erased, placing erased in front of a parameter list (like given).

```
def methodWithErasedEv(erased ev: Ev): Int = 42
val lambdaWithErasedEv: erased Ev => Int =
   (erased ev: Ev) => 42
```

erased parameters will not be usable for computations, though they can be used as arguments to other erased parameters.

```
def methodWithErasedInt1(erased i: Int): Int =
   i + 42 // ERROR: can not use i

def methodWithErasedInt2(erased i: Int): Int =
   methodWithErasedInt1(i) // OK
```

Not only parameters can be marked as erased, **val** and **def** can also be marked with **erased**. These will also only be usable as arguments to **erased** parameters.

```
erased val erasedEvidence: Ev = ...
methodWithErasedEv(erasedEvidence)
```

5.7.3 What happens with erased values at runtime?

As erased are guaranteed not to be used in computations, they can and will be erased.

```
// becomes def methodWithErasedEv(): Int at runtime
def methodWithErasedEv(erased ev: Ev): Int = ...

def evidence1: Ev = ...
erased def erasedEvidence2: Ev = ... // does not exist at runtime
erased val erasedEvidence3: Ev = ... // does not exist at runtime

// evidence1 is not evaluated and no value is passed to methodWithErasedEv
methodWithErasedEv(evidence1)
```

5.7.4 State machine with erased evidence example

The following example is an extended implementation of a simple state machine which can be in a state On or Off. The machine can change state from Off to On with turnedOn only if it is currently Off, conversely from On to Off with turnedOff only if it is currently On. These last constraint are captured with the IsOff[S] and IsOn[S] given evidence only exist for IsOff[Off] and IsOn[On]. For example, not allowing calling turnedOff on in an Off state as we would require an evidence IsOn[Off] that will not be found.

As the given evidences of turnedOn and turnedOff are not used in the bodies of those functions we can mark them as erased. This will remove the evidence parameters at runtime, but we would still evaluate the isOn and isOff givens that were found as arguments. As isOn and isOff are not used except as erased arguments, we can mark them as erased, hence removing the evaluation of the isOn and isOff evidences.

```
import scala.annotation.implicitNotFound

sealed trait State
final class On extends State
final class Off extends State

@implicitNotFound("State must be Off")
class IsOff[S <: State]
object IsOff:
    // will not be called at runtime for turnedOn, the
    // compiler will only require that this evidence exists
    given IsOff[Off] = new IsOff[Off]

@implicitNotFound("State must be On")
class IsOn[S <: State]
object IsOn:
    // will not exist at runtime, the compiler will only</pre>
```

```
// require that this evidence exists at compile time
   erased given IsOn[On] = new IsOn[On]
class Machine[S <: State] private ():</pre>
   // ev will disappear from both functions
   def turnedOn(using erased ev: IsOff[S]): Machine[On] = new Machine[On]
   def turnedOff(using erased ev: IsOn[S]): Machine[Off] = new Machine[Off]
object Machine:
   def newMachine(): Machine[Off] = new Machine[Off]
@main def test =
   val m = Machine.newMachine()
  m.turnedOn
  m.turnedOn.turnedOff
   // m.turnedOff
   //
                 State must be On
   // m.turnedOn.turnedOn
   //
   //
                          State must be Off
Note that in Inline we discussed erasedValue and inline matches. erasedValue is
implemented with erased, so the state machine above can be encoded as follows:
import scala.compiletime._
sealed trait State
final class On extends State
final class Off extends State
class Machine[S <: State]:</pre>
   transparent inline def turnOn(): Machine[On] =
      inline erasedValue[S] match
         case : Off => new Machine[On]
         case _: On => error("Turning on an already turned on machine")
   transparent inline def turnOff(): Machine[Off] =
      inline erasedValue[S] match
         case _: On => new Machine[Off]
         case _: Off => error("Turning off an already turned off machine")
object Machine:
   def newMachine(): Machine[Off] =
      println("newMachine")
```

```
new Machine[Off]
end Machine

@main def test =
    val m = Machine.newMachine()
    m.turnOn()
    m.turnOn().turnOff()
    m.turnOn().turnOff()
    m.turnOn().turnOn() // error: Turning on an already turned on machine
```

More Details

5.8 The Meta-theory of Symmetric Metaprogramming

This note presents a simplified variant of principled metaprogramming and sketches its soundness proof. The variant treats only dialogues between two stages. A program can have quotes which can contain splices (which can contain quotes, which can contain splices, and so on). Or the program could start with a splice with embedded quotes. The essential restriction is that (1) a term can contain top-level quotes or top-level splices, but not both, and (2) quotes cannot appear directly inside quotes and splices cannot appear directly inside splices. In other words, the universe is restricted to two phases only.

Under this restriction we can simplify the typing rules so that there are always exactly two environments instead of having a stack of environments. The variant presented here differs from the full calculus also in that we replace evaluation contexts with contextual typing rules. While this is more verbose, it makes it easier to set up the meta theory.

5.8.1 Syntax

```
Terms
                   ::=
                                                 variable
                           (x: T) \Rightarrow t
                                                 lambda
                                                 application
                                                 quote
                           ~t
                                                 splice
Simple terms
                          x \mid (x: T) \Rightarrow u \mid u u
                     ::=
                           (x: T) \Rightarrow t
Values
                     ::=
                                                 lambda
                                                 quoted value
Types
                 Τ
                   ::=
                                                 base type
                           Α
                           T -> T
                                                 function type
                           'nТ
                                                 quoted type
```

5.8.2 Operational semantics

5.8.2.1 Evaluation

5.8.2.2 Splicing

5.8.3 Typing Rules

Typing judgments are of the form $E1 * E2 \mid - t$: Twhere E1, E2 are environments and * is one of ~ and '.

(Curiously, this looks a bit like a Christmas tree).

5.8.4 Soundness

The meta-theory typically requires mutual inductions over two judgments.

E1 ' E2 |- ~t: T

5.8.4.1 Progress Theorem

- 1. If E1 \sim |- t: T then either t = v for some value v or t --> t2 for some term t2
- 2. If 'E2 \mid t: T then either t = u for some simple term u or t ==> t2 for some term t2.

Proof by structural induction over terms.

To prove (1):

- the cases for variables, lambdas and applications are as in STLC.
- If t = 't2, then by inversion we have 'E1 |- t2: T2 for some type T2. By the second induction hypothesis (I.H.), we have one of:
 - t2 = u, hence 't2 is a value,
 - t2 ==> t3, hence 't2 --> 't3.
- The case t = -t2 is not typable.

To prove (2):

- If t = x then t is a simple term.
- If $t = (x: T) \Rightarrow t2$, then either t2 is a simple term, in which case t is as well. Or by the second I.H. $t2 \Longrightarrow t3$, in which case $t \Longrightarrow (x: T) \Longrightarrow t3$.
- If t = t1 t2 then one of three cases applies:
 - t1 and t2 are a simple term, then t is as well a simple term.

- t1 is not a simple term. Then by the second I.H., t1 ==> t12, hence t ==> t12 t2.
- t1 is a simple term but t2 is not. Then by the second I.H. t2 ==> t22, hence t ==> t1 t22.
- The case t = 't2 is not typable.
- If t = ~t2 then by inversion we have E2 ~ |-t2: ~T2, for some type T2. By the first I.H., we have one of
 - t2 = v. Since t2: 'T2, we must have v = 'u, for some simple term u, hence t = ~'u. By quote-splice reduction, t ==> u.
 - t2 --> t3. Then by the context rule for 't, t ==> 't3.

5.8.4.2 Substitution Lemma

- 1. If E1 ~ E2 \mid s: S and E1 ~ E2, x: S \mid t: T then E1 ~ E2 \mid \mid x := s]t: T.
- 2. If E1 ~ E2 |- s: S and E2, x: S ' E1 |- t: T then E2 ' E1 |- [x := s]t: T.

The proofs are by induction on typing derivations for t, analogous to the proof for STL (with (2) a bit simpler than (1) since we do not need to swap lambda bindings with the bound variable x). The arguments that link the two hypotheses are as follows.

To prove (1), let t = 't1. Then T = 'T1 for some type T1 and the last typing rule is

By the second I.H. E2 ' E1 |-[x := s]t1: T1. By typing, E1 ~ E2 |- '[x := s]t1: 'T1. Since [x := s]t = [x := s]('t1) = '[x := s]t1 we get [x := s]t: 'T1.

To prove (2), let t = -t1. Then the last typing rule is

By the first I.H., E1 ~ E2 |- [x := s]t1: 'T. By typing, E2 ' E1 |- ~ [x := s]t1: T. Since [x := s]t = [x := s](~t1) = ~ [x := s]t1 we get [x := s]t: T.

5.8.4.3 Preservation Theorem

- 1. If E1 ~ E2 \mid t1: T and t1 --> t2 then E1 ~ E2 \mid t2: T.
- 2. If E1 ' E2 \mid t1: T and t1 ==> t2 then E1 ' E2 \mid t2: T.

The proof is by structural induction on evaluation derivations. The proof of (1) is analogous to the proof for STL, using the substitution lemma for the beta reduction case, with the addition of reduction of quoted terms, which goes as follows:

• Assume the last rule was

By inversion of typing rules, we must have T = 'T1 for some type T1 such that t1: T1. By the second I.H., t2: T1, hence 't2:T1'.

To prove (2):

• Assume the last rule was ~'u ==> u. The typing proof of ~'u must have the form

Hence, E1 ' E2 |- u: T.

• Assume the last rule was

By typing inversion, E1 ' E2, x: S \mid - t1: T1 for some type T1 such that T = S -> T1. By the I.H, t2: T1. By the typing rule for lambdas the result follows.

- The context rules for applications are equally straightforward.
- Assume the last rule was

By inversion of typing rules, we must have t1: 'T. By the first I.H., t2: 'T, hence ~t2: T.

6 Other New Features

6.1 Trait Parameters

Scala 3 allows traits to have parameters, just like classes have parameters.

```
trait Greeting(val name: String):
    def msg = s"How are you, $name"

class C extends Greeting("Bob"):
    println(msg)
```

Arguments to a trait are evaluated immediately before the trait is initialized.

One potential issue with trait parameters is how to prevent ambiguities. For instance, you might try to extend **Greeting** twice, with different parameters.

```
class D extends C, Greeting("Bill") // error: parameter passed twice
```

Should this print "Bob" or "Bill"? In fact this program is illegal, because it violates the second rule of the following for trait parameters:

- 1. If a class C extends a parameterized trait T, and its superclass does not, C *must* pass arguments to T.
- 2. If a class C extends a parameterized trait T, and its superclass does as well, C must not pass arguments to T.
- 3. Traits must never pass arguments to parent traits.

Here's a trait extending the parameterized trait Greeting.

```
trait FormalGreeting extends Greeting:
   override def msg = s"How do you do, $name"
```

As is required, no arguments are passed to Greeting. However, this poses an issue when defining a class that extends FormalGreeting:

```
class E extends FormalGreeting // error: missing arguments for `Greeting`.
```

The correct way to write E is to extend both Greeting and FormalGreeting (in either order):

```
class E extends Greeting("Bob"), FormalGreeting
```

6.1.1 Reference

For more information, see Scala SIP 25.

6.2 Transparent Traits

Traits are used in two roles:

1. As mixing for other classes and traits

2. As types of vals, defs, or parameters

Some traits are used primarily in the first role, and we usually do not want to see them in inferred types. An example is the Product trait that the compiler adds as a mixin trait to every case class or case object. In Scala 2, this parent trait sometimes makes inferred types more complicated than they should be. Example:

```
trait Kind
case object Var extends Kind
case object Val extends Kind
val x = Set(if condition then Val else Var)
```

Here, the inferred type of x is Set[Kind & Product & Serializable] whereas one would have hoped it to be Set[Kind]. The reasoning for this particular type to be inferred is as follows:

- The type of the conditional above is the union type Val | Var.
- A union type is widened in type inference to the least supertype that is not a union type. In the example, this type is Kind & Product & Serializable since all three traits are traits of both Val and Var. So that type becomes the inferred element type of the set.

Scala 3 allows one to mark a mixin trait as transparent, which means that it can be suppressed in type inference. Here's an example that follows the lines of the code above, but now with a new transparent trait S instead of Product:

```
transparent trait S trait Kind object Var extends Kind, S object Val extends Kind, S val x = Set(if condition then Val else Var)
```

Now x has inferred type Set[Kind]. The common transparent trait S does not appear in the inferred type.

6.2.1 Transparent Traits

The traits scala. Product, java.lang. Serializable and java.lang. Comparable are treated automatically as transparent. Other traits are turned into transparent traits using the modifier transparent. Scala 2 traits can also be made transparent by adding a @transparentTrait annotation. This annotation is defined in scala.annotation. It will be deprecated and phased out once Scala 2/3 interopability is no longer needed.

Typically, transparent traits are traits that influence the implementation of inheriting classes and traits that are not usually used as types by themselves. Two examples from the standard collection library:

- IterableOps, which provides method implementations for an Iterable
- StrictOptimizedSeqOps, which optimises some of these implementations for sequences with efficient indexing.

Generally, any trait that is extended recursively is a good candidate to be declared transparent.

6.2.2 Rules for Inference

Transparent traits can be given as explicit types as usual. But they are often elided when types are inferred. Roughly, the rules for type inference say that transparent traits are dropped from intersections where possible.

The precise rules are as follows:

- When inferring a type of a type variable, or the type of a val, or the return type of a def,
- where that type is not higher-kinded,
- and where B is its known upper bound or Any if none exists:
- If the type inferred so far is of the form T1 & ... & Tn where n >= 1, replace the maximal number of transparent Tis by Any, while ensuring that the resulting type is still a subtype of the bound B.
- However, do not perform this widening if all transparent traits Ti can get replaced in that way.

The last clause ensures that a single transparent trait instance such as Product is not widened to Any. Transparent trait instances are only dropped when they appear in conjunction with some other type.

6.3 Universal Apply Methods

Scala case classes generate apply methods, so that values of case classes can be created using simple function application, without needing to write new.

Scala 3 generalizes this scheme to all concrete classes. Example:

```
class StringBuilder(s: String):
    def this() = this("")

StringBuilder("abc") // same as new StringBuilder("abc")
StringBuilder() // same as new StringBuilder()
```

This works since a companion object with two apply methods is generated together with the class. The object looks like this:

```
object StringBuilder:
   inline def apply(s: String): StringBuilder = new StringBuilder(s)
   inline def apply(): StringBuilder = new StringBuilder()
```

The synthetic object StringBuilder and its apply methods are called *constructor* proxies. Constructor proxies are generated even for Java classes and classes coming from Scala 2. The precise rules are as follows:

1. A constructor proxy companion object object C is created for a concrete class C, provided the class does not have already a companion, and there is also no

other value or method named C defined or inherited in the scope where C is defined.

- 2. Constructor proxy apply methods are generated for a concrete class provided
 - the class has a companion object (which might have been generated in step 1), and
 - that companion object does not already define a member named apply.

Each generated apply method forwards to one constructor of the class. It has the same type and value parameters as the constructor.

Constructor proxy companions cannot be used as values by themselves. A proxy companion object must be selected with apply (or be applied to arguments, in which case the apply is implicitly inserted).

Constructor proxies are also not allowed to shadow normal definitions. That is, if an identifier resolves to a constructor proxy, and the same identifier is also defined or imported in some other scope, an ambiguity is reported.

6.3.0.1 Motivation Leaving out new hides an implementation detail and makes code more pleasant to read. Even though it requires a new rule, it will likely increase the perceived regularity of the language, since case classes already provide function call creation syntax (and are often defined for this reason alone).

6.4 Export clauses

An export clause defines aliases for selected members of an object. Example:

```
class BitMap
class InkJet

class Printer:
    type PrinterType
    def print(bits: BitMap): Unit = ???
    def status: List[String] = ???

class Scanner:
    def scan(): BitMap = ???
    def status: List[String] = ???

class Copier:
    private val printUnit = new Printer { type PrinterType = InkJet }
    private val scanUnit = new Scanner

    export scanUnit.scan
    export printUnit.{status => _, _}

    def status: List[String] = printUnit.status ++ scanUnit.status
```

The two export clauses define the following export aliases in class Copier:

```
final def scan(): BitMap = scanUnit.scan()
final def print(bits: BitMap): Unit = printUnit.print(bits)
final type PrinterType = printUnit.PrinterType
```

They can be accessed inside Copier as well as from outside:

```
val copier = new Copier
copier.print(copier.scan())
```

An export clause has the same format as an import clause. Its general form is:

```
export path . { sel_1, ..., sel_n }
export given path . { sel_1, ..., sel_n }
```

It consists of a qualifier expression path, which must be a stable identifier, followed by one or more selectors sel_i that identify what gets an alias. Selectors can be of one of the following forms:

- A *simple selector* **x** creates aliases for all eligible members of **path** that are named **x**.
- A renaming selector $x \Rightarrow y$ creates aliases for all eligible members of path that are named x, but the alias is named y instead of x.
- An omitting selector $x \Rightarrow$ prevents x from being aliased by a subsequent wildcard selector.
- A wildcard selector creates aliases for all eligible members of path except for synthetic members generated by the compiler and those members that are named by a previous simple, renaming, or omitting selector.

A member is *eliqible* if all of the following holds:

- its owner is not a base class of the class(*) containing the export clause,
- the member does not override a concrete definition that has as owner a base class of the class containing the export clause.
- it is accessible at the export clause,
- it is not a constructor, nor the (synthetic) class part of an object,
- it is a given instance (or an old-style implicit value) if and only if the export is tagged with given.

It is a compile-time error if a simple or renaming selector does not identify any eligible members.

Type members are aliased by type definitions, and term members are aliased by method definitions. Export aliases copy the type and value parameters of the members they refer to. Export aliases are always final. Aliases of given instances are again defined as givens (and aliases of old-style implicits are implicit). Aliases of inline methods or values are again defined inline. There are no other modifiers that can be given to an alias. This has the following consequences for overriding:

• Export aliases cannot be overridden, since they are final.

- Export aliases cannot override concrete members in base classes, since they are not marked override.
- However, export aliases can implement deferred members of base classes.

Export aliases for public value definitions that are accessed without referring to private values in the qualifier path are marked by the compiler as "stable" and their result types are the singleton types of the aliased definitions. This means that they can be used as parts of stable identifier paths, even though they are technically methods. For instance, the following is OK:

```
class C { type T }
object O { val c: C = ... }
export O.c
def f: c.T = ...
```

Export clauses can appear in classes or they can appear at the top-level. An export clause cannot appear as a statement in a block.

(*) Note: Unless otherwise stated, the term "class" in this discussion also includes object and trait definitions.

6.4.1 Motivation

It is a standard recommendation to prefer composition over inheritance. This is really an application of the principle of least power: Composition treats components as blackboxes whereas inheritance can affect the internal workings of components through overriding. Sometimes the close coupling implied by inheritance is the best solution for a problem, but where this is not necessary the looser coupling of composition is better.

So far, object-oriented languages including Scala made it much easier to use inheritance than composition. Inheritance only requires an extends clause whereas composition required a verbose elaboration of a sequence of forwarders. So in that sense, object-oriented languages are pushing programmers to a solution that is often too powerful. Export clauses redress the balance. They make composition relationships as concise and easy to express as inheritance relationships. Export clauses also offer more flexibility than extends clauses since members can be renamed or omitted.

Export clauses also fill a gap opened by the shift from package objects to top-level definitions. One occasionally useful idiom that gets lost in this shift is a package object inheriting from some class. The idiom is often used in a facade like pattern, to make members of internal compositions available to users of a package. Top-level definitions are not wrapped in a user-defined object, so they can't inherit anything. However, top-level definitions can be export clauses, which supports the facade design pattern in a safer and more flexible way.

6.4.2 Syntax changes:

```
TemplateStat ::= ...
```

6.4.3 Elaboration of Export Clauses

Export clauses raise questions about the order of elaboration during type checking. Consider the following example:

```
class B { val c: Int }
object a { val b = new B }
export a._
export b._
```

Is the export b._ clause legal? If yes, what does it export? Is it equivalent to export a.b._? What about if we swap the last two clauses?

```
export b._
export a._
```

To avoid tricky questions like these, we fix the elaboration order of exports as follows.

Export clauses are processed when the type information of the enclosing object or class is completed. Completion so far consisted of the following steps:

- 1. Elaborate any annotations of the class.
- 2. Elaborate the parameters of the class.
- 3. Elaborate the self type of the class, if one is given.
- 4. Enter all definitions of the class as class members, with types to be completed on demand.
- 5. Determine the types of all parents of the class.

With export clauses, the following steps are added:

- 6. Compute the types of all paths in export clauses in a context logically inside the class but not considering any imports or exports in that class.
- 7. Enter export aliases for the eligible members of all paths in export clauses.

It is important that steps 6 and 7 are done in sequence: We first compute the types of *all* paths in export clauses and only after this is done we enter any export aliases as class members. This means that a path of an export clause cannot refer to an alias made available by another export clause of the same class.

6.5 Opaque Type Aliases

Opaque types aliases provide type abstraction without any overhead. Example:

```
object Logarithms:
    opaque type Logarithm = Double

object Logarithm:

    // These are the two ways to lift to the Logarithm type

def apply(d: Double): Logarithm = math.log(d)

def safe(d: Double): Option[Logarithm] =
    if d > 0.0 then Some(math.log(d)) else None

end Logarithm

// Extension methods define opaque types' public APIs
    extension (x: Logarithm)
    def toDouble: Double = math.exp(x)
    def + (y: Logarithm): Logarithm = Logarithm(math.exp(x) + math.exp(y))
    def * (y: Logarithm): Logarithm = x + y
```

end Logarithms

This introduces Logarithm as a new abstract type, which is implemented as Double. The fact that Logarithm is the same as Double is only known in the scope where Logarithm is defined which in the above example corresponds to the object Logarithms. Or in other words, within the scope it is treated as type alias, but this is opaque to the outside world where in consequence Logarithm is seen as an abstract type and has nothing to do with Double.

The public API of Logarithm consists of the apply and safe methods defined in the companion object. They convert from Doubles to Logarithm values. Moreover, an operation toDouble that converts the other way, and operations + and * are defined as extension methods on Logarithm values. The following operations would be valid because they use functionality implemented in the Logarithms object.

import Logarithms.Logarithm

```
val 1 = Logarithm(1.0)
val 12 = Logarithm(2.0)
val 13 = 1 * 12
val 14 = 1 + 12
```

But the following operations would lead to type errors:

```
val d: Double = 1  // error: found: Logarithm, required: Double
val 12: Logarithm = 1.0 // error: found: Double, required: Logarithm
1 * 2  // error: found: Int(2), required: Logarithm
1 // error: `/` is not a member of Logarithm
```

6.5.0.1 Bounds For Opaque Type Aliases Opaque type aliases can also come with bounds. Example:

object Access:

```
opaque type Permissions = Int
opaque type PermissionChoice = Int
opaque type Permission <: Permissions & PermissionChoice = Int
extension (x: Permissions)
  def & (y: Permissions): Permissions = x | y
extension (x: PermissionChoice)
   def | (y: PermissionChoice): PermissionChoice = x | y
extension (granted: Permissions)
   def is(required: Permissions) = (granted & required) == required
extension (granted: Permissions)
   def isOneOf(required: PermissionChoice) = (granted & required) != 0
val NoPermission: Permission = 0
val Read: Permission = 1
val Write: Permission = 2
val ReadWrite: Permissions = Read | Write
val ReadOrWrite: PermissionChoice = Read | Write
```

end Access

The Access object defines three opaque type aliases:

- Permission, representing a single permission,
- Permissions, representing a set of permissions with the meaning "all of these permissions granted",
- PermissionChoice, representing a set of permissions with the meaning "at least one of these permissions granted".

Outside the Access object, values of type Permissions may be combined using the & operator, where x & y means "all permissions in x and in y granted". Values of type PermissionChoice may be combined using the | operator, where $x \mid y$ means "a permission in x or in y granted".

Note that inside the Access object, the & and | operators always resolve to the corresponding methods of Int, because members always take precedence over extension methods. Because of that, the | extension method in Access does not cause infinite recursion. Also, the definition of ReadWrite must use |, even though an equivalent definition outside Access would use &.

All three opaque type aliases have the same underlying representation type Int. The Permission type has an upper bound Permissions & PermissionChoice. This makes it known outside the Access object that Permission is a subtype of the other two types. Hence, the following usage scenario type-checks.

```
object User:
   import Access._

case class Item(rights: Permissions)

val roItem = Item(Read)  // OK, since Permission <: Permissions
val rwItem = Item(ReadWrite)
val noItem = Item(NoPermission)

assert(!roItem.rights.is(ReadWrite))
assert(roItem.rights.isOneOf(ReadOrWrite))

assert(rwItem.rights.is(ReadWrite))
assert(rwItem.rights.isOneOf(ReadOrWrite))

assert(!noItem.rights.is(ReadWrite))
assert(!noItem.rights.is(ReadWrite))
assert(!noItem.rights.is(ReadWrite))
end User</pre>
```

On the other hand, the call roltem.rights.isOneOf(ReadWrite) would give a type error since Permissions and PermissionChoice are different, unrelated types outside Access.

More details

6.6 Open Classes

An open modifier on a class signals that the class is planned for extensions. Example:

```
// File Writer.scala
package p

open class Writer[T]:
    /** Sends to stdout, can be overridden */
    def send(x: T) = println(x)

    /** Sends all arguments using `send` */
    def sendAll(xs: T*) = xs.foreach(send)
end Writer

// File EncryptedWriter.scala
package p

class EncryptedWriter[T: Encryptable] extends Writer[T]:
    override def send(x: T) = super.send(encrypt(x))
```

An open class typically comes with some documentation that describes the internal

calling patterns between methods of the class as well as hooks that can be overridden. We call this the *extension contract* of the class. It is different from the *external contract* between a class and its users.

Classes that are not open can still be extended, but only if at least one of two alternative conditions is met:

- The extending class is in the same source file as the extended class. In this case, the extension is usually an internal implementation matter.
- The language feature adhocExtensions is enabled for the extending class. This is typically enabled by an import clause in the source file of the extension:

```
import scala.language.adhocExtensions
```

Alternatively, the feature can be enabled by the compiler option—language:adhocExtensions. If the feature is not enabled, the compiler will issue a "feature" warning. For instance, if the open modifier on class Writer is dropped, compiling EncryptedWriter would produce a warning:

```
-- Feature Warning: EncryptedWriter.scala:6:14 ----
|class EncryptedWriter[T: Encryptable] extends Writer[T]
|
|Unless class Writer is declared 'open', its extension
| in a separate file should be enabled
|by adding the import clause 'import scala.language.adhocExtensions'
|or by setting the compiler option -language:adhocExtensions.
```

6.6.0.1 Motivation When writing a class, there are three possible expectations of extensibility:

- 1. The class is intended to allow extensions. This means one should expect a carefully worked out and documented extension contract for the class.
- 2. Extensions of the class are forbidden, for instance to make correctness or security guarantees.
- 3. There is no firm decision either way. The class is not a priori intended for extensions, but if others find it useful to extend on an ad-hoc basis, let them go ahead. However, they are on their own in this case. There is no documented extension contract, and future versions of the class might break the extensions (by rearranging internal call patterns, for instance).

The three cases are clearly distinguished by using open for (1), final for (2) and no modifier for (3).

It is good practice to avoid *ad-hoc* extensions in a code base, since they tend to lead to fragile systems that are hard to evolve. But there are still some situations where these extensions are useful: for instance, to mock classes in tests, or to apply temporary patches that add features or fix bugs in library classes. That's why *ad-hoc* extensions are permitted, but only if there is an explicit opt-in via a language feature import.

6.6.0.2 Details

- open is a soft modifier. It is treated as a normal identifier unless it is in modifier position.
- An open class cannot be final or sealed.
- Traits or abstract classes are always open, so open is redundant for them.

6.6.0.3 Relationship with sealed A class that is neither abstract nor open is similar to a sealed class: it can still be extended, but only in the same compilation unit. The difference is what happens if an extension of the class is attempted in another compilation unit. For a sealed class, this is an error, whereas for a simple non-open class, this is still permitted provided the adhocExtensions feature is enabled, and it gives a warning otherwise.

6.6.0.4 Migration open is a new modifier in Scala 3. To allow cross compilation between Scala 2.13 and Scala 3.0 without warnings, the feature warning for ad-hoc extensions is produced only under -source 3.1. It will be produced by default from Scala 3.1 on.

6.7 Parameter Untupling

Say you have a list of pairs

```
val xs: List[(Int, Int)]
```

and you want to map **xs** to a list of **Ints** so that each pair of numbers is mapped to their sum. Previously, the best way to do this was with a pattern-matching decomposition:

```
xs map {
   case (x, y) => x + y
}
```

While correct, this is also inconvenient and confusing, since the case suggests that the pattern match could fail. As a shorter and clearer alternative Scala 3 now allows

```
xs.map {
    (x, y) => x + y
}
```

or, equivalently:

```
xs.map(_ + _)
```

Generally, a function value with n > 1 parameters is converted to a pattern-matching closure using case if the expected type is a unary function type of the form $((T_1, \ldots, T_n)) \Rightarrow U$.

6.7.1 Reference

For more information see:

- More details
- Issue #897.

6.8 Kind Polymorphism

Normally type parameters in Scala are partitioned into *kinds*. First-level types are types of values. Higher-kinded types are type constructors such as List or Map. The kind of a type is indicated by the top type of which it is a subtype. Normal types are subtypes of Any, covariant single argument type constructors such as List are subtypes of [+X] =>> Any, and the Map type constructor is a subtype of [X, +Y] =>> Any.

A type can be used only as prescribed by its kind. Subtypes of Any cannot be applied to type arguments whereas subtypes of [X] =>> Any must be applied to a type argument, unless they are passed to type parameters of the same kind.

Sometimes we would like to have type parameters that can have more than one kind, for instance to define an implicit value that works for parameters of any kind. This is now possible through a form of (*subtype*) kind polymorphism. Kind polymorphism relies on the special type scala. AnyKind that can be used as an upper bound of a type.

```
def f[T <: AnyKind] = ...</pre>
```

The actual type arguments of **f** can then be types of arbitrary kinds. So the following would all be legal:

```
f[Int]
f[List]
f[Map]
f[[X] =>> String]
```

We call type parameters and abstract types with an AnyKind upper bound any-kinded types. Since the actual kind of an any-kinded type is unknown, its usage must be heavily restricted: An any-kinded type can be neither the type of a value, nor can it be instantiated with type parameters. So about the only thing one can do with an any-kinded type is to pass it to another any-kinded type argument. Nevertheless, this is enough to achieve some interesting generalizations that work across kinds, typically through advanced uses of implicits.

(todo: insert good concise example)

Some technical details: AnyKind is a synthesized class just like Any, but without any members. It extends no other class. It is declared abstract and final, so it can be neither instantiated nor extended.

AnyKind plays a special role in Scala's subtype system: It is a supertype of all other types no matter what their kind is. It is also assumed to be kind-compatible with all other types. Furthermore, AnyKind is treated as a higher-kinded type (so it cannot be used as a type of values), but at the same time it has no type parameters (so it cannot be instantiated).

Note: This feature is considered experimental but stable and it can be disabled under compiler flag (i.e. -Yno-kind-polymorphism).

6.9 The Matchable Trait

A new trait Matchable controls the ability to pattern match.

6.9.1 The Problem

The Scala 3 standard library has a type IArray for immutable arrays that is defined like this:

```
opaque type IArray[+T] = Array[ <: T]
```

The IArray type offers extension methods for length and apply, but not for update; hence it seems values of type IArray cannot be updated.

However, there is a potential hole due to pattern matching. Consider:

```
val imm: IArray[Int] = ...
imm match
  case a: Array[Int] => a(0) = 1
```

The test will succeed at runtime since IArrays are represented as Arrays at runtime. But if we allowed it, it would break the fundamental abstraction of immutable arrays.

Aside: One could also achieve the same by casting:

```
imm.asInstanceOf[Array[Int]](0) = 1
```

But that is not as much of a problem since in Scala asInstanceOf is understood to be low-level and unsafe. By contrast, a pattern match that compiles without warning or error should not break abstractions.

Note also that the problem is not tied to opaque types as match selectors. The following slight variant with a value of parametric type T as match selector leads to the same problem:

```
def f[T](x: T) = x match
   case a: Array[Int] => a(0) = 0
f(imm)
```

Finally, note that the problem is not linked to just opaque types. No unbounded type parameter or abstract type should be decomposable with a pattern match.

6.9.2 The Solution

There is a new type scala. Matchable that controls pattern matching. When typing a pattern match of a constructor pattern C(...) or a type pattern _: C it is required that the selector type conforms to Matchable. If that's not the case a warning is issued. For instance when compiling the example at the start of this section we get:

To allow migration from Scala 2 and cross-compiling between Scala 2 and 3 the warning is turned on only for -source 3.1-migration or higher.

Matchable is a universal trait with Any as its parent class. It is extended by both AnyVal and AnyRef. Since Matchable is a supertype of every concrete value or reference class it means that instances of such classes can be matched as before. However, match selectors of the following types will produce a warning:

- Type Any: if pattern matching is required one should use Matchable instead.
- Unbounded type parameters and abstract types: If pattern matching is required they should have an upper bound Matchable.
- Type parameters and abstract types that are only bounded by some universal trait: Again, Matchable should be added as a bound.

Here is the hierarchy of top-level classes and traits with their defined methods:

```
abstract class Any:
    def getClass
    def isInstanceOf
    def asInstanceOf
    def ==
    def !=
    def ##
    def equals
    def hashCode
    def toString

trait Matchable extends Any

class AnyVal extends Any, Matchable
class Object extends Any, Matchable
```

Matchable is currently a marker trait without any methods. Over time we might migrate methods getClass and isInstanceOf to it, since these are closely related to pattern-matching.

6.10 OthreadUnsafe Annotation

A new annotation **@threadUnsafe** can be used on a field which defines a lazy val. When this annotation is used, the initialization of the lazy val will use a faster mechanism which is not thread-safe.

6.10.0.1 Example

```
import scala.annotation.threadUnsafe
class Hello:
    @threadUnsafe lazy val x: Int = 1
```

6.11 OtargetName Annotations

A @targetName annotation on a definition defines an alternate name for the implementation of that definition. Example:

def ++= [T] (ys: Vec[T]): Vec[T] = ...

Here, the ++= operation is implemented (in Byte code or native code) under the name append. The implementation name affects the code that is generated, and is the name under which code from other languages can call the method. For instance, ++= could be invoked from Java like this:

```
VecOps.append(vec1, vec2)
```

The @targetName annotation has no bearing on Scala usages. Any application of that method in Scala has to use ++=, not append.

6.11.0.1 Details

- 1. @targetName is defined in package scala.annotation. It takes a single argument of type String. That string is called the *external name* of the definition that's annotated.
- 2. A @targetName annotation can be given for all kinds of definitions.
- 3. The name given in a @targetName annotation must be a legal name for the defined entities on the host platform.
- 4. It is recommended that definitions with symbolic names have a @targetName annotation. This will establish an alternate name that is easier to search for and will avoid cryptic encodings in runtime diagnostics.
- 5. Definitions with names in backticks that are not legal host platform names should also have a <code>@targetName</code> annotation.

6.11.0.2 Relationship with Overriding @targetName annotations are significant for matching two method definitions to decide whether they conflict or override each other. Two method definitions match if they have the same name, signature, and erased name. Here,

- The *signature* of a definition consists of the names of the erased types of all (value-) parameters and the method's result type.
- The *erased name* of a method definition is its target name if a <code>@targetName</code> annotation is given and its defined name otherwise.

This means that **@targetName** annotations can be used to disambiguate two method definitions that would otherwise clash. For instance.

```
def f(x: => String): Int = x.length
def f(x: => Int): Int = x + 1 // error: double definition
```

The two definitions above clash since their erased parameter types are both Function0, which is the type of the translation of a by-name-parameter. Hence they have the same names and signatures. But we can avoid the clash by adding a <code>@targetName</code> annotation to either method or to both of them. Example:

```
@targetName("f_string")
def f(x: => String): Int = x.length
def f(x: => Int): Int = x + 1 // OK
```

This will produce methods f_string and f in the generated code.

However, <code>@targetName</code> annotations are not allowed to break overriding relationships between two definitions that have otherwise the same names and types. So the following would be in error:

```
import annotation.targetName
class A:
    def f(): Int = 1
class B extends A:
    @targetName("g") def f(): Int = 2
```

The compiler reports here:

The relevant overriding rules can be summarized as follows:

- Two members can override each other if their names and signatures are the same, and they either have the same erased names or the same types.
- If two members override, then both their erased names and their types must be the same.

As usual, any overriding relationship in the generated code must also be present in the original code. So the following example would also be in error:

```
import annotation.targetName
class A:
```

```
def f(): Int = 1
class B extends A:
    @targetName("f") def g(): Int = 2
```

Here, the original methods g and f do not override each other since they have different names. But once we switch to target names, there is a clash that is reported by the compiler:

6.12 New Control Syntax

Scala 3 has a new "quiet" syntax for control expressions that does not rely on enclosing the condition in parentheses, and also allows to drop parentheses or braces around the generators of a for-expression. Examples:

```
if x < 0 then
   "negative"
else if x == 0 then
   "zero"
else
   "positive"
if x < 0 then -x else x
while x >= 0 do x = f(x)
for x \leftarrow xs if x > 0
yield x * x
for
   x <- xs
   y <- ys
do
   println(x + y)
try body
catch case ex: IOException => handle
```

The rules in detail are:

• The condition of an if-expression can be written without enclosing parenthe-

ses if it is followed by a then.

- The condition of a while-loop can be written without enclosing parentheses if it is followed by a do.
- The enumerators of a for-expression can be written without enclosing parentheses or braces if they are followed by a yield or do.
- A do in a for-expression expresses a for-loop.
- A catch can be followed by a single case on the same line. If there are multiple cases, these have to appear within braces (just like in Scala 2) or an indented block. ### Rewrites

The Scala 3 compiler can rewrite source code from old syntax to new syntax and back. When invoked with options -rewrite -new-syntax it will rewrite from old to new syntax, dropping parentheses and braces in conditions and enumerators. When invoked with options -rewrite -old-syntax it will rewrite in the reverse direction, inserting parentheses and braces as needed.

6.13 Optional Braces

As an experimental feature, Scala 3 enforces some rules on indentation and allows some occurrences of braces {...} to be optional. It can be turned off with the compiler flag -noindent.

- First, some badly indented programs are flagged with warnings.
- Second, some occurrences of braces {...} are made optional. Generally, the rule is that adding a pair of optional braces will not change the meaning of a well-indented program.

6.13.0.1 Indentation Rules The compiler enforces two rules for well-indented programs, flagging violations as warnings.

1. In a brace-delimited region, no statement is allowed to start to the left of the first statement after the opening brace that starts a new line.

This rule is helpful for finding missing closing braces. It prevents errors like:

```
if (x < 0) {
  println(1)
  println(2)

println("done") // error: indented too far to the left</pre>
```

2. If significant indentation is turned off (i.e. under Scala 2 mode or under -noindent) and we are at the start of an indented sub-part of an expression, and the indented part ends in a newline, the next statement must start at an indentation width less than the sub-part. This prevents errors where an opening brace was forgotten, as in

```
if (x < 0)
  println(1)
  println(2) // error: missing `{`</pre>
```

These rules still leave a lot of leeway how programs should be indented. For instance, they do not impose any restrictions on indentation within expressions, nor do they require that all statements of an indentation block line up exactly.

The rules are generally helpful in pinpointing the root cause of errors related to missing opening or closing braces. These errors are often quite hard to diagnose, in particular in large programs.

6.13.0.2 Optional Braces The compiler will insert <indent> or <outdent> tokens at certain line breaks. Grammatically, pairs of <indent> and <outdent> tokens have the same effect as pairs of braces { and }.

The algorithm makes use of a stack IW of previously encountered indentation widths. The stack initially holds a single element with a zero indentation width. The *current indentation width* is the indentation width of the top of the stack.

There are two rules:

- 1. An <indent> is inserted at a line break, if
 - An indentation region can start at the current position in the source, and
 - the first token on the next line has an indentation width strictly greater than the current indentation width

An indentation region can start

- after the leading parameters of an extension, or
- after a with in a given instance, or
- after a ": at end of line" token (see below)
- after one of the following tokens:

```
= => <- catch do else finally for
if match return then try while yield</pre>
```

If an <indent> is inserted, the indentation width of the token on the next line is pushed onto IW, which makes it the new current indentation width.

- 2. An **<outdent>** is inserted at a line break, if
 - the first token on the next line has an indentation width strictly less than the current indentation width, and
 - the last token on the previous line is not one of the following tokens which indicate that the previous statement continues:

```
then else do catch finally yield match
```

• the first token on the next line is not a leading infix operator.

If an **<outdent>** is inserted, the top element is popped from IW. If the indentation width of the token on the next line is still less than the new current

indentation width, step (2) repeats. Therefore, several **<outdent>** tokens may be inserted in a row.

An <outdent> is also inserted if the next token following a statement sequence starting with an <indent> closes an indentation region, i.e. is one of then, else, do, catch, finally, yield, },),] or case.

An <outdent> is finally inserted in front of a comma that follows a statement sequence starting with an <indent> if the indented region is itself enclosed in parentheses

It is an error if the indentation width of the token following an **<outdent>** does not match the indentation of some previous line in the enclosing indentation region. For instance, the following would be rejected.

```
if x < 0 then
     -x
else // error: `else` does not align correctly
    x</pre>
```

Indentation tokens are only inserted in regions where newline statement separators are also inferred: at the top-level, inside braces {...}, but not inside parentheses (...), patterns or types.

6.13.0.3 Optional Braces Around Template Bodies The Scala grammar uses the term *template body* for the definitions of a class, trait, or object that are normally enclosed in braces. The braces around a template body can also be omitted by means of the following rule

If at the point where a template body can start there is a: that occurs at the end of a line, and that is followed by at least one indented statement, the recognized token is changed from ":" to ": at end of line". The latter token is one of the tokens that can start an indentation region. The Scala grammar is changed so an optional ": at end of line" is allowed in front of a template body.

Analogous rules apply for enum bodies and local packages containing nested definitions.

With these new rules, the following constructs are all valid:

```
trait A:
    def f: Int

class C(x: Int) extends A:
    def f = x

object O:
    def f = 3

enum Color:
    case Red, Green, Blue
```

```
new A:
    def f = 3

package p:
    def a = 1

package q:
    def b = 2
```

In each case, the : at the end of line can be replaced without change of meaning by a pair of braces that enclose the following indented definition(s).

The syntax changes allowing this are as follows:

```
Template ::= InheritClauses [colonEol] [TemplateBody]
EnumDef ::= id ClassConstr InheritClauses [colonEol] EnumBody
Packaging ::= 'package' QualId [nl | colonEol] '{' TopStatSeq '}'
SimpleExpr ::= 'new' ConstrApp {'with' ConstrApp} [[colonEol] TemplateBody]
```

Here, colonEol stands for ": at end of line", as described above. The lexical analyzer is modified so that a : at the end of a line is reported as colonEol if the parser is at a point where a colonEol is valid as next token.

6.13.0.4 Spaces vs Tabs Indentation prefixes can consist of spaces and/or tabs. Indentation widths are the indentation prefixes themselves, ordered by the string prefix relation. So, so for instance "2 tabs, followed by 4 spaces" is strictly less than "2 tabs, followed by 5 spaces", but "2 tabs, followed by 4 spaces" is incomparable to "6 tabs" or to "4 spaces, followed by 2 tabs". It is an error if the indentation width of some line is incomparable with the indentation width of the region that's current at that point. To avoid such errors, it is a good idea not to mix spaces and tabs in the same source file.

6.13.0.5 Indentation and Braces Indentation can be mixed freely with braces. For interpreting indentation inside braces, the following rules apply.

- 1. The assumed indentation width of a multiline region enclosed in braces is the indentation width of the first token that starts a new line after the opening brace.
- 2. On encountering a closing brace }, as many <outdent> tokens as necessary are inserted to close all open indentation regions inside the pair of braces.

6.13.0.6 Special Treatment of Case Clauses The indentation rules for match expressions and catch clauses are refined as follows:

• An indentation region is opened after a match or catch also if the following case appears at the indentation width that's current for the match itself.

• In that case, the indentation region closes at the first token at that same indentation width that is not a case, or at any token with a smaller indentation width, whichever comes first.

The rules allow to write match expressions where cases are not indented themselves, as in the example below:

```
x match
    case 1 => print("I")
    case 2 => print("II")
    case 3 => print("III")
    case 4 => print("IV")
    case 5 => print("V")
```

6.13.0.7 The End Marker Indentation-based syntax has many advantages over other conventions. But one possible problem is that it makes it hard to discern when a large indentation region ends, since there is no specific token that delineates the end. Braces are not much better since a brace by itself also contains no information about what region is closed.

To solve this problem, Scala 3 offers an optional end marker. Example:

An end marker consists of the identifier end and a follow-on specifier token that together constitute all the tokes of a line. Possible specifier tokens are identifiers or one of the following keywords

```
if while for match try new this val given
```

End markers are allowed in statement sequences. The specifier token **s** of an end marker must correspond to the statement that precedes it. This means:

- If the statement defines a member x then s must be the same identifier x.
- If the statement defines a constructor then s must be this.
- If the statement defines an anonymous given, then s must be given.
- If the statement defines an anonymous extension, then s must be extension.
- If the statement defines an anonymous class, then s must be new.
- If the statement is a val definition binding a pattern, then s must be val.
- If the statement is a package clause that refers to package p, then s must be the same identifier p.

• If the statement is an if, while, for, try, or match statement, then s must be that same token.

For instance, the following end markers are all legal:

```
package p1.p2:
```

```
abstract class C():
   def this(x: Int) =
      this()
      if x > 0 then
         val a :: b =
            x :: Nil
         end val
         var y =
            Х
         end y
         while y > 0 do
            println(y)
            y -= 1
         end while
         try
            x match
               case 0 => println("0")
               case =>
            end match
         finally
            println("done")
         end try
      end if
   end this
   def f: String
end C
object C:
   given C =
      new C:
         def f = "!"
         end f
      end new
   end given
end C
extension (x: C)
   def ff: String = x.f ++ x.f
```

end extension

end p2

6.13.0.7.1 When to Use End Markers It is recommended that end markers are used for code where the extent of an indentation region is not immediately apparent "at a glance". People will have different preferences what this means, but one can nevertheless give some guidelines that stem from experience. An end marker makes sense if

- the construct contains blank lines, or
- the construct is long, say 15-20 lines or more,
- the construct ends heavily indented, say 4 indentation levels or more.

If none of these criteria apply, it's often better to not use an end marker since the code will be just as clear and more concise. If there are several ending regions that satisfy one of the criteria above, we usually need an end marker only for the outermost closed region. So cascades of end markers as in the example above are usually better avoided.

6.13.0.7.2 Syntax

6.13.0.8 Example Here is a (somewhat meta-circular) example of code using indentation. It provides a concrete representation of indentation widths as defined above together with efficient operations for constructing and comparing indentation widths.

enum IndentWidth:

```
def < (that: IndentWidth): Boolean =</pre>
      this <= that && !(that <= this)
  override def toString: String =
     this match
      case Run(ch, n) =>
         val kind = ch match
            case ' ' => "space"
            case '\t' => "tab"
            case => s"'$ch'-character"
         val suffix = if n == 1 then "" else "s"
         s"$n $kind$suffix"
      case Conc(1, r) =>
         s"$1, $r"
object IndentWidth:
  private inline val MaxCached = 40
  private val spaces = IArray.tabulate(MaxCached + 1)(new Run(' ', '))
  private val tabs = IArray.tabulate(MaxCached + 1)(new Run('\t', _))
  def Run(ch: Char, n: Int): Run =
      if n <= MaxCached && ch == ' ' then
         spaces(n)
      else if n <= MaxCached && ch == '\t' then
         tabs(n)
      else
        new Run(ch, n)
  end Run
  val Zero = Run(' ', 0)
end IndentWidth
```

6.13.0.9 Settings and Rewrites Significant indentation is enabled by default. It can be turned off by giving any of the options -noindent, old-syntax and language:Scala2. If indentation is turned off, it is nevertheless checked that indentation conforms to the logical program structure as defined by braces. If that is not the case, the compiler issues a warning.

The Scala 3 compiler can rewrite source code to indented code and back. When invoked with options -rewrite -indent it will rewrite braces to indented regions where possible. When invoked with options -rewrite -noindent it will rewrite in the reverse direction, inserting braces for indentation regions. The -indent option only works on new-style syntax. So to go from old-style syntax to new-style indented code one has to invoke the compiler twice, first with options -rewrite -new-syntax, then again with options -rewrite -indent. To go in the opposite direction, from indented code to old-style syntax, it's -rewrite -noindent, followed by -rewrite

-old-syntax.

6.13.0.10 Variant: Indentation Marker: Generally, the possible indentation regions coincide with those regions where braces {...} are also legal, no matter whether the braces enclose an expression or a set of definitions. There is one exception, though: Arguments to function can be enclosed in braces but they cannot be simply indented instead. Making indentation always significant for function arguments would be too restrictive and fragile.

To allow such arguments to be written without braces, a variant of the indentation scheme is implemented under option <code>-Yindent-colons</code>. This variant is more contentious and less stable than the rest of the significant indentation scheme. In this variant, a colon: at the end of a line is also one of the possible tokens that opens an indentation region. Examples:

```
times(10):
    println("ah")
    println("ha")

or

xs.map:
    x =>
    val y = x - 1
    y * y
```

The colon is usable not only for lambdas and by-name parameters, but also even for ordinary parameters:

```
credentials ++ :
  val file = Path.userHome / ".credentials"
  if file.exists
  then Seq(Credentials(file))
  else Seq()
```

How does this syntax variant work? Colons at the end of lines are their own token, distinct from normal:. The Scala grammar is changed so that colons at end of lines are accepted at all points where an opening brace enclosing an argument is legal. Special provisions are taken so that method result types can still use a colon on the end of a line, followed by the actual type on the next.

6.14 Explicit Nulls

Explicit nulls is an opt-in feature that modifies the Scala type system, which makes reference types (anything that extends AnyRef) non-nullable.

This means the following code will no longer typecheck:

```
val x: String = null // error: found `Null`, but required `String`
Instead, to mark a type as nullable we use a union type
```

```
val x: String | Null = null // ok
```

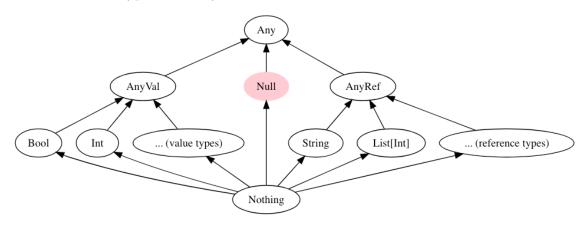
Explicit nulls are enabled via a -Yexplicit-nulls flag.

Read on for details.

6.14.1 New Type Hierarchy

When explicit nulls are enabled, the type hierarchy changes so that Null is only a subtype of Any, as opposed to every reference type.

This is the new type hierarchy:



After erasure, Null remains a subtype of all reference types (as forced by the JVM).

6.14.2 Unsoundness

The new type system is unsound with respect to null. This means there are still instances where an expression has a non-nullable type like String, but its value is actually null.

The unsoundness happens because uninitialized fields in a class start out as null:

```
class C:
```

```
val f: String = foo(f)
def foo(f2: String): String = f2

val c = new C()
// c.f == "field is null"
```

The unsoundness above can be caught by the compiler with the option -Ycheck-init. More details can be found in safe initialization.

6.14.3 Equality

We don't allow the double-equal (== and !=) and reference (eq and ne) comparison between AnyRef and Null anymore, since a variable with a non-nullable type cannot have null as value. null can only be compared with Null, nullable union (T | Null), or Any type.

For some reason, if we really want to compare null with non-null values, we have to provide a type hint (e.g.: Any).

6.14.4 Working with Null

To make working with nullable values easier, we propose adding a few utilities to the standard library. So far, we have found the following useful:

• An extension method .nn to "cast away" nullability

```
def[T] (x: T|Null) nn: x.type & T =
   if x == null then new NullPointerException("tried to cast away nullability,
   else x.asInstanceOf[x.type & T]
```

This means that given x: String | Null, x.nn has type String, so we can call all the usual methods on it. Of course, x.nn will throw a NPE if x is null.

Don't use .nn on mutable variables directly, because it may introduce an unknown type into the type of the variable.

6.14.5 Java Interoperability

The Scala compiler can load Java classes in two ways: from source or from bytecode. In either case, when a Java class is loaded, we "patch" the type of its members to reflect that Java types remain implicitly nullable.

Specifically, we patch * the type of fields * the argument type and return type of methods

UncheckedNull is an alias for Null with magic properties (see below). We illustrate the rules with following examples:

• The first two rules are easy: we nullify reference types but not value types.

```
class C {
   String s;
   int x;
}
```

```
==>
class C:
  val s: String|UncheckedNull
  val x: Int
```

• We nullify type parameters because in Java a type parameter is always nullable, so the following code compiles.

```
class C<T> { T foo() { return null; } }
==>
class C[T] { def foo(): T|UncheckedNull }
```

Notice this is rule is sometimes too conservative, as witnessed by

```
class InScala:
```

```
val c: C[Bool] = ??? // C as above
val b: Bool = c.foo() // no longer typechecks, since foo now returns Bool/
```

• This reduces the number of redundant nullable types we need to add. Consider

```
class Box<T> { T get(); }
class BoxFactory<T> { Box<T> makeBox(); }
==>
class Box[T] { def get(): T|UncheckedNull }
class BoxFactory[T] { def makeBox(): Box[T]|UncheckedNull }
```

Suppose we have a BoxFactory [String]. Notice that calling makeBox() on it returns a Box [String] | UncheckedNull, not a Box [String | UncheckedNull] | UncheckedNull. This seems at first glance unsound ("What if the box itself has null inside?"), but is sound because calling get() on a Box [String] returns a String | UncheckedNull.

Notice that we need to patch *all* Java-defined classes that transitively appear in the argument or return type of a field or method accessible from the Scala code being compiled. Absent crazy reflection magic, we think that all such Java classes *must* be visible to the Typer in the first place, so they will be patched.

• We will append UncheckedNull to the type arguments if the generic class is defined in Scala.

```
class BoxFactory<T> {
   Box<T> makeBox(); // Box is Scala-defined
   List<Box<List<T>>> makeCrazyBoxes(); // List is Java-defined
}
==>
```

```
class BoxFactory[T]:
    def makeBox(): Box[T | UncheckedNull] | UncheckedNull
    def makeCrazyBoxes(): List[Box[List[T] | UncheckedNull]] | UncheckedNull
```

In this case, since Box is Scala-defined, we will get Box [T|UncheckedNull] | UncheckedNull. This is needed because our nullability function is only applied (modularly) to the Java classes, but not to the Scala ones, so we need a way to tell Box that it contains a nullable value.

The List is Java-defined, so we don't append UncheckedNull to its type argument. But we still need to nullify its inside.

• We don't nullify *simple* literal constant (final) fields, since they are known to be non-null

```
class Constants {
    final String NAME = "name";
    final int AGE = 0;
    final char CHAR = 'a';

    final String NAME_GENERATED = getNewName();
}

==>
class Constants:
    val NAME: String("name") = "name"
    val AGE: Int(0) = 0
    val CHAR: Char('a') = 'a'

val NAME_GENERATED: String | Null = ???
```

• We don't append UncheckedNull to a field nor to a return type of a method which is annotated with a NotNull annotation.

```
class C {
    @NotNull String name;
    @NotNull List<String> getNames(String prefix); // List is Java-defined
    @NotNull Box<String> getBoxedName(); // Box is Scala-defined
}
==>
class C:
    val name: String
    def getNames(prefix: String | UncheckedNull): List[String] // we still need
    def getBoxedName(): Box[String | UncheckedNull] // we don't append `UncheckedNull]
```

The annotation must be from the list below to be recognized as NotNull by the compiler. Check Definitions.scala for an updated list.

```
// A list of annotations that are commonly used to indicate
```

```
// that a field/method argument or return type is not null.
// These annotations are used by the nullification logic in
// JavaNullInterop to improve the precision of type nullification.
// We don't require that any of these annotations be present
// in the class path, but we want to create Symbols for the
// ones that are present, so they can be checked during nullification.
@tu lazy val NotNullAnnots: List[ClassSymbol] = ctx.getClassesIfDefined(
  "javax.annotation.Nonnull" ::
 "edu.umd.cs.findbugs.annotations.NonNull" ::
 "androidx.annotation.NonNull" ::
  "android.support.annotation.NonNull" ::
 "android.annotation.NonNull" ::
 "com.android.annotations.NonNull" ::
 "org.eclipse.jdt.annotation.NonNull" ::
  "org.checkerframework.checker.nullness.qual.NonNull" ::
 "org.checkerframework.checker.nullness.compatqual.NonNullDecl" ::
 "org.jetbrains.annotations.NotNull" ::
 "lombok.NonNull" ::
 "io.reactivex.annotations.NonNull" :: Nil map PreNamedString)
```

6.14.5.1 UncheckedNull To enable method chaining on Java-returned values, we have the special type alias for Null:

```
type UncheckedNull = Null
```

UncheckedNull behaves just like Null, except it allows (unsound) member selections:

```
// Assume someJavaMethod()'s original Java signature is
// String someJavaMethod() {}
val s2: String = someJavaMethod().trim().substring(2).toLowerCase() // unsound
```

Here, all of trim, substring and toLowerCase return a String|UncheckedNull. The Typer notices the UncheckedNull and allows the member selection to go through. However, if someJavaMethod were to return null, then the first member selection would throw a NPE.

Without UncheckedNull, the chaining becomes too cumbersome

```
val ret = someJavaMethod()
val s2 =
  if ret != null then
    val tmp = ret.trim()
    if tmp != null then
      val tmp2 = tmp.substring(2)
      if tmp2 != null then
            tmp2.toLowerCase()
// Additionally, we need to handle the `else` branches.
```

6.14.6 Flow Typing

We added a simple form of flow-sensitive type inference. The idea is that if p is a stable path or a trackable variable, then we can know that p is non-null if it's compared with null. This information can then be propagated to the then and else branches of an if-statement (among other places).

Example:

```
val s: String|Null = ???
if s != null then
   // s: String
// s: String | Null
assert(x != null)
// s: String
A similar inference can be made for the else case if the test is p == null
if s == null then
   // s: String/Null
else
   // s: String
== and != is considered a comparison for the purposes of the flow inference.
6.14.6.1 Logical Operators We also support logical operators (&&, ||, and !):
val s: String|Null = ???
val s2: String|Null = ???
if s != null \&\& s2 != null then
   // s: String
   // s2: String
if s == null \mid \mid s2 == null then
   // s: String | Null
   // s2: String|Null
else
   // s: String
   // s2: String
```

6.14.6.2 Inside Conditions We also support type specialization *within* the condition, taking into account that && and || are short-circuiting:

```
val s: String|Null = ???

if s != null && s.length > 0 then // s: String in `s.length > 0`
    // s: String
```

```
if s == null || s.length > 0 then // s: String in `s.length > 0`
    // s: String | Null
else
    // s: String
```

6.14.6.3 Match Case The non-null cases can be detected in match statements.

```
val s: String|Null = ????
s match
  case _: String => // s: String
  case _ =>
```

6.14.6.4 Mutable Variable We are able to detect the nullability of some local mutable variables. A simple example is:

```
class C(val x: Int, val next: C|Null)

var xs: C|Null = C(1, C(2, null))

// xs is trackable, since all assignments are in the same method

while xs != null do

// xs: C

val xsx: Int = xs.x

val xscpy: C = xs

xs = xscpy // since xscpy is non-null, xs still has type C after this line

// xs: C

xs = xs.next // after this assignment, xs can be null again

// xs: C | Null
```

When dealing with local mutable variables, there are two questions:

1. Whether to track a local mutable variable during flow typing. We track a local mutable variable iff the variable is not assigned in a closure. For example, in the following code \mathbf{x} is assigned to by the closure \mathbf{y} , so we do not do flow typing on \mathbf{x} .

```
var x: String|Null = ???
def y =
    x = null

if x != null then
    // y can be called here, which would break the fact
    val a: String = x // error: x is captured and mutated by the closure, not
```

2. Whether to generate and use flow typing on a specific use of a local mutable variable. We only want to do flow typing on a use that belongs to the same method as the definition of the local variable. For example, in the following code, even x is not assigned to by a closure, but we can only use flow typing in

one of the occurrences (because the other occurrence happens within a nested closure).

```
var x: String|Null = ???
def y =
   if x != null then
      // not safe to use the fact (x != null) here
      // since y can be executed at the same time as the outer block
      val _: String = x
if x != null then
   val a: String = x // ok to use the fact here
   x = null
```

See more examples in tests/explicit-nulls/neg/var-ref-in-closure.scala.

Currently, we are unable to track paths with a mutable variable prefix. For example, x.a if x is mutable.

6.14.6.5 Unsupported Idioms We don't support:

- flow facts not related to nullability (if x == 0 then { // x: 0.type not inferred })
- tracking aliasing between non-nullable paths

```
val s: String|Null = ???
val s2: String|Null = ???
if s != null && s == s2 then
    // s: String inferred
    // s2: String not inferred
```

6.14.7 Binary Compatibility

Our strategy for binary compatibility with Scala binaries that predate explicit nulls and new libraries compiled without **-Yexplicit-nulls** is to leave the types unchanged and be compatible but unsound.

More details

6.15 Safe Initialization

Scala 3 implements experimental safe initialization check, which can be enabled by the compiler option -Ycheck-init.

6.15.1 A Quick Glance

To get a feel of how it works, we first show several examples below.

6.15.1.1 Parent-Child Interaction Given the following code snippet:

```
abstract class AbstractFile:
   def name: String
   val extension: String = name.substring(4)
class RemoteFile(url: String) extends AbstractFile:
   val localFile: String = s"${url.##}.tmp" // error: usage of `localFile` before
   def name: String = localFile
The checker will report:
-- Warning: tests/init/neg/AbstractFile.scala:7:4 ------
7 | val localFile: String = s"${url.##}.tmp" // error: usage of `localFile
      Access non-initialized field value localFile. Calling trace:
       -> val extension: String = name.substring(4) [ AbstractFile.scala:3
         -> def name: String = localFile
                                                          [ AbstractFile.scala:8
6.15.1.2 Inner-Outer Interaction Given the code below:
object Trees:
   class ValDef { counter += 1 }
   class EmptyValDef extends ValDef
   val theEmptyValDef = new EmptyValDef
   private var counter = 0 // error
The checker will report:
-- Warning: tests/init/neg/trees.scala:5:14 ------
5 | private var counter = 0 // error
               Access non-initialized field variable counter. Calling trace:
                -> val theEmptyValDef = new EmptyValDef [ trees.scala:4 ] 
-> class EmptyValDef extends ValDef [ trees.scala:3 ]
                  -> class ValDef { counter += 1 }
                                                              [trees.scala:2]
6.15.1.3 Functions Given the code below:
abstract class Parent:
   val f: () => String = () => this.message
   def message: String
class Child extends Parent:
   val a = f()
   val b = "hello"  // error
   def message: String = b
The checker reports:
-- Warning: tests/init/neg/features-high-order.scala:7:6 -----
7 | val b = "hello"
                             // error
```

6.15.2 Design Goals

We establish the following design goals:

- **Sound**: checking always terminates, and is sound for common and reasonable usage (over-approximation)
- Expressive: support common and reasonable initialization patterns
- Friendly: simple rules, minimal syntactic overhead, informative error messages
- Modular: modular checking, no analysis beyond project boundary
- Fast: instant feedback
- Simple: no changes to core type system, explainable by a simple theory

By reasonable usage, we include the following use cases (but not restricted to them):

- Access fields on this and outer this during initialization
- Call methods on this and outer this during initialization
- Instantiate inner class and call methods on such instances during initialization
- Capture fields in functions

6.15.3 Principles

To achieve the goals, we uphold three fundamental principles: *stackability*, *monotonicity* and *scopability*.

Stackability means that all fields of a class are initialized at the end of the class body. Scala enforces this property in syntax by demanding that all fields are initialized at the end of the primary constructor, except for the language feature below:

```
var x: T =
```

Control effects such as exceptions may break this property, as the following example shows:

```
class MyException(val b: B) extends Exception("")
class A:
   val b = try { new B } catch { case myEx: MyException => myEx.b }
   println(b.a)

class B:
   throw new MyException(this)
   val a: Int = 1
```

In the code above, the control effect teleport the uninitialized value wrapped in an exception. In the implementation, we avoid the problem by ensuring that the values

that are thrown must be transitively initialized.

Monotonicity means that the initialization status of an object should not go backward: initialized fields continue to be initialized, a field points to an initialized object may not later point to an object under initialization. As an example, the following code will be rejected:

In the code above, suppose ctx points to a transitively initialized object. Now the assignment at line 3 makes this, which is not fully initialized, reachable from ctx. This makes field usage dangerous, as it may indirectly reach uninitialized fields.

Monotonicity is based on a well-known technique called *heap monotonic typestate* to ensure soundness in the presence of aliasing [1]. Roughly, it means initialization state should not go backwards.

Scopability means that access to partially constructed objects should be controlled by static scoping. Control effects like coroutines, delimited control, resumable exceptions may break the property, as they can transport a value upper in the stack (not in scope) to be reachable from the current scope. Static fields can also serve as a teleport thus breaks this property. In the implementation, we need to enforce that teleported values are transitively initialized.

The principles enable *local reasoning* of initialization, which means:

An initialized environment can only produce initialized values.

For example, if the arguments to an new-expression are transitively initialized, so is the result. If the receiver and arguments in a method call are transitively initialized, so is the result.

6.15.4 Rules

With the established principles and design goals, following rules are imposed:

1. In an assignment o.x = e, the expression e may only point to transitively initialized objects.

This is how monotonicity is enforced in the system. Note that in an initialization val f: T = e, the expression e may point to an object under initialization. This requires a distinction between mutation and initialization in order to enforce different rules. Scala has different syntax for them, it thus is not an issue.

2. References to objects under initialization may not be passed as arguments to method calls or constructors.

Escape of this in the constructor is commonly regarded as an anti-pattern, and it's rarely used in practice. This rule is simple for the programmer to reason about initialization and it simplifies implementation. The theory supports safe escape of this with the help of annotations, we delay the extension until there is a strong need.

3. Local definitions may only refer to transitively initialized objects.

It means that in a local definition $\mathtt{val}\ \mathtt{x}\colon \mathtt{T}=\mathtt{e},$ the expression \mathtt{e} may only evaluate to transitively initialized objects. The same goes for local lazy variables and methods. This rule is again motivated for simplicity in reasoning about initialization: programmers may safely assume that all local definitions only point to transitively initialized objects.

6.15.5 Modularity (considered)

Currently, the analysis works across project boundaries based on TASTy. The following is a proposal to make the checking more modular. The feedback from the community is welcome.

For modularity, we need to forbid subtle initialization interaction beyond project boundaries. For example, the following code passes the check when the two classes are defined in the same project:

```
class Base:
```

```
private val map: mutable.Map[Int, String] = mutable.Map.empty
def enter(k: Int, v: String) = map(k) = v

class Child extends Base:
  enter(1, "one")
  enter(2, "two")
```

However, when the class Base and Child are defined in two different projects, the check can emit a warning for the calls to enter in the class Child. This restricts subtle initialization within project boundaries, and avoids accidental violation of contracts across library versions.

We can impose the following rules to enforce modularity:

- 1. A class or trait that may be extended in another project should not call *virtual* methods on this in its template/mixin evaluation, directly or indirectly.
- 2. The method call o.m(args) is forbidden if o is not transitively initialized and the target of m is defined in an external project.
- 3. The expression new p.C(args) is forbidden, if p is not transitively initialized and C is defined in an external project.

6.15.6 Theory

The theory is based on type-and-effect systems [2]. We introduce two concepts, effects and potentials:

```
= this | Warm(C, ) | .f | .m | .super[D] | Cold | Fun(\Pi, \Phi) | .outer[C] = \uparrow | .f! | .m!
```

Potentials () represent values that are possibly under initialization.

- this: current object
- Warm(C,): an object of type C where all its fields are assigned, and the potential for this of its enclosing class is .
- .f: the potential of the field f in the potential
- .m: the potential of the field f in the potential
- .super[D]: essentially the object , used for virtual method resolution
- Cold: an object with unknown initialization status
- Fun(Π , Φ): a function, when called produce effects Φ and return potentials Π .
- ullet .outer[C]: the potential of this for the enclosing class of C when C.this is

Effects are triggered from potentials:

- 1: promote the object pointed to by the potential to fully-initialized
- .f!: access field f on the potential
- .m!: call the method m on the potential

To ensure that the checking always terminate and for better performance, we restrict the length of potentials to be finite (by default 2). If the potential is too long, the checker stops tracking it by checking that the potential is actually transitively initialized.

For an expression e, it may be summarized by the pair (Π, Φ) , which means evaluation of e may produce the effects Φ and return the potentials Π . Each field and method is associated with such a pair. We call such a pair *summary*. The expansion of proxy potentials and effects, such as .f, .m and .m!, will take advantage of the summaries. Depending on the potential for this, the summaries need to be rebased (asSeenFrom) before usage.

The checking treats the templates of concrete classes as entry points. It maintains the set of initialized fields as initialization progresses, and check that only initialized fields are accessed during the initialization and there is no leaking of values under initialization. Virtual method calls on this is not a problem, as they can always be resolved statically.

6.15.7 Back Doors

Occasionally you may want to suppress warnings reported by the checker. You can either write e: @unchecked to tell the checker to skip checking for the expression e, or you may use the old trick: mark some fields as lazy.

6.15.8 Caveats

The system cannot handle static fields, nor does it provide safety guarantee when extending Java or Scala 2 classes. Calling methods of Java or Scala 2 is always safe.

6.15.9 References

- Fähndrich, M. and Leino, K.R.M., 2003, July. *Heap monotonic typestates*. In International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO).
- Lucassen, J.M. and Gifford, D.K., 1988, January. *Polymorphic effect systems*. In Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (pp. 47-57). ACM.

6.16 Named Type Arguments

Note: This feature is implemented in Scala 3, but is not expected to be part of Scala 3.0.

Type arguments of methods can now be specified by name as well as by position. Example:

```
def construct[Elem, Coll[_]](xs: Elem*): Coll[Elem] = ???
val xs1 = construct[Coll = List, Elem = Int](1, 2, 3)
val xs2 = construct[Coll = List](1, 2, 3)
```

Similar to a named value argument (x = e), a named type argument [X = T] instantiates the type parameter X to the type T. Named type arguments do not have to be in order (see xs1 above) and unspecified arguments are inferred by the compiler (see xs2 above). Type arguments must be all named or un-named, mixtures of named and positional type arguments are not supported.

6.16.1 Motivation

The main benefit of named type arguments is that unlike positional arguments, you are allowed to omit passing arguments for some parameters, like in the definition of xs2 above. A missing type argument is inferred as usual by local type inference. This is particularly useful in situations where some type arguments can be easily inferred from others.

More details

6.17 TypeTest ☑

6.17.1 TypeTest

When pattern matching there are two situations where a runtime type test must be performed. The first kind is an explicit type test using the ascription pattern notation.

```
(x: X) match case y: Y =>
```

The second is when an extractor takes an argument that is not a subtype of the scrutinee type.

```
(x: X) match
  case y @ Y(n) =>

object Y:
  def unapply(x: Y): Some[Int] = ...
```

In both cases, a class test will be performed at runtime. But when the type test is on an abstract type (type parameter or type member), the test cannot be performed because the type is erased at runtime.

A TypeTest can be provided to make this test possible.

```
package scala.reflect

trait TypeTest[-S, T]:
   def unapply(s: S): Option[s.type & T]
```

It provides an extractor that returns its argument typed as a T if the argument is a T. It can be used to encode a type test.

```
def f[X, Y](x: X)(using tt: TypeTest[X, Y]): Option[Y] = x match
  case tt(x @ Y(1)) => Some(x)
  case tt(x) => Some(x)
  case _ => None
```

To avoid the syntactic overhead the compiler will look for a type test automatically if it detects that the type test is on abstract types. This means that x: Y is transformed to tt(x) and $x @ Y(_)$ to $tt(x @ Y(_))$ if there is a contextual TypeTest[X, Y] in scope. The previous code is equivalent to

```
def f[X, Y](x: X)(using TypeTest[X, Y]): Option[Y] = x match
  case x @ Y(1) => Some(x)
  case x: Y => Some(x)
  case => None
```

We could create a type test at call site where the type test can be performed with runtime class tests directly as follows

```
val tt: TypeTest[Any, String] =
  new TypeTest[Any, String]:
    def unapply(s: Any): Option[s.type & String] = s match
        case s: String => Some(s)
        case _ => None

f[AnyRef, String]("acb")(using tt)
```

The compiler will synthesize a new instance of a type test if none is found in scope as:

```
new TypeTest[A, B]:
  def unapply(s: A): Option[s.type & B] = s match
    case s: B => Some(s)
    case _ => None
```

If the type tests cannot be done there will be an unchecked warning that will be raised on the case $s: B \Rightarrow test$.

The most common TypeTest instances are the ones that take any parameters (i.e. TypeTest[Any, T]). To make it possible to use such instances directly in context bounds we provide the alias

```
package scala.reflect

type Typeable[T] = TypeTest[Any, T]

This alias can be used as

def f[T: Typeable]: Boolean =
   "abc" match
      case x: T => true
      case _ => false

f[String] // true
f[Int] // false
```

6.17.1.1 TypeTest and ClassTag TypeTest is a replacement for functionality provided previously by ClassTag.unapply. Using ClassTag instances was unsound since classtags can check only the class component of a type. TypeTest fixes that unsoundness. ClassTag type tests are still supported but a warning will be emitted after 3.0.

6.17.2 Examples

Given the following abstract definition of Peano numbers that provides TypeTest[Nat, Zero] and TypeTest[Nat, Succ]

```
trait Peano:
   type Nat
   type Zero <: Nat
   type Succ <: Nat

def safeDiv(m: Nat, n: Succ): (Nat, Nat)

val Zero: Zero

val Succ: SuccExtractor</pre>
```

```
trait SuccExtractor:
      def apply(nat: Nat): Succ
      def unapply(nat: Succ): Option[Nat]
   given TypeTest[Nat, Zero] = typeTestOfZero
   protected def typeTestOfZero: TypeTest[Nat, Zero]
   given TypeTest[Nat, Succ] = typeTestOfSucc
   protected def typeTestOfSucc: TypeTest[Nat, Succ]
it will be possible to write the following program
val peano: Peano = ...
import peano._
def divOpt(m: Nat, n: Nat): Option[(Nat, Nat)] =
      case Zero => None
      case s @ Succ(_) => Some(safeDiv(m, s))
val two = Succ(Succ(Zero))
val five = Succ(Succ(Succ(two)))
println(divOpt(five, two))
Note that without the TypeTest[Nat, Succ] the pattern Succ.unapply(nat:
Succ) would be unchecked.
```

7 Other Changed Features

7.1 Numeric Literals

Note: This feature is not yet part of the Scala 3 language definition. It can be made available by a language import:

```
import scala.language.experimental.genericNumberLiterals
```

In Scala 2, numeric literals were confined to the primitive numeric types Int, Long, Float, and Double. Scala 3 allows to write numeric literals also for user-defined types. Example:

```
val x: Long = -10_000_000_000
val y: BigInt = 0x123_abc_789_def_345_678_901
val z: BigDecimal = 110_222_799_799.99

(y: BigInt) match
    case 123_456_789_012_345_678_901 =>
```

The syntax of numeric literals is the same as before, except there are no pre-set limits how large they can be.

7.1.0.1 Meaning of Numeric Literals The meaning of a numeric literal is determined as follows:

- If the literal ends with 1 or L, it is a Long integer (and must fit in its legal range).
- If the literal ends with f or F, it is a single precision floating point number of type Float.
- If the literal ends with d or D, it is a double precision floating point number of type Double.

In each of these cases the conversion to a number is exactly as in Scala 2 or in Java. If a numeric literal does *not* end in one of these suffixes, its meaning is determined by the expected type:

- 1. If the expected type is Int, Long, Float, or Double, the literal is treated as a standard literal of that type.
- 2. If the expected type is a fully defined type T that has a given instance of type scala.util.FromDigits[T], the literal is converted to a value of type T by passing it as an argument to the fromDigits method of that instance (more details below).
- 3. Otherwise, the literal is treated as a Double literal (if it has a decimal point or an exponent), or as an Int literal (if not). (This last possibility is again as in Scala 2 or Java.)

With these rules, the definition

```
val x: Long = -10_000_000_000
```

is legal by rule (1), since the expected type is Long. The definitions

```
val y: BigInt = 0x123_abc_789_def_345_678_901
val z: BigDecimal = 111222333444.55
```

are legal by rule (2), since both BigInt and BigDecimal have FromDigits instances (which implement the FromDigits subclasses FromDigits.WithRadix and FromDigits.Decimal, respectively). On the other hand,

```
val x = -10 000 000 000
```

gives a type error, since without an expected type -10_000_000_000 is treated by rule (3) as an Int literal, but it is too large for that type.

7.1.0.2 The FromDigits Trait To allow numeric literals, a type simply has to define a given instance of the scala.util.FromDigits type class, or one of its subclasses. FromDigits is defined as follows:

```
trait FromDigits[T]:
   def fromDigits(digits: String): T
```

Implementations of the fromDigits convert strings of digits to the values of the implementation type T. The digits string consists of digits between 0 and 9, possibly preceded by a sign ("+" or "-"). Number separator characters _ are filtered out before the string is passed to fromDigits.

The companion object FromDigits also defines subclasses of FromDigits for whole numbers with a given radix, for numbers with a decimal point, and for numbers that can have both a decimal point and an exponent:

object FromDigits:

```
/** A subclass of `FromDigits` that also allows to convert whole
    number literals with a radix other than 10
    */
trait WithRadix[T] extends FromDigits[T]:
    def fromDigits(digits: String): T = fromDigits(digits, 10)
    def fromDigits(digits: String, radix: Int): T

/** A subclass of `FromDigits` that also allows to convert number
    * literals containing a decimal point ".".
    */
trait Decimal[T] extends FromDigits[T]

/** A subclass of `FromDigits` that allows also to convert number
    * literals containing a decimal point "." or an
    * exponent `('e' / 'E')['+' / '-']digit digit*`.
    */
trait Floating[T] extends Decimal[T]
```

A user-defined number type can implement one of those, which signals to the compiler that hexadecimal numbers, decimal points, or exponents are also accepted in literals for this type.

7.1.0.3 Error Handling FromDigits implementations can signal errors by throwing exceptions of some subtype of FromDigitsException. FromDigitsException is defined with three subclasses in the FromDigits object as follows:

```
abstract class FromDigitsException(msg: String) extends NumberFormatException(msg)

class NumberTooLarge (msg: String = "number too large") extends FromDigitsE

class NumberTooSmall (msg: String = "number too small") extends FromDigitsE
```

class MalformedNumber(msg: String = "malformed number literal") extends FromDigitsF

7.1.0.4 Example As a fully worked out example, here is an implementation of a new numeric class, BigFloat, that accepts numeric literals. BigFloat is defined in terms of a BigInt mantissa and an Int exponent:

```
case class BigFloat(mantissa: BigInt, exponent: Int):
   override def toString = s"${mantissa}e${exponent}"
```

BigFloat literals can have a decimal point as well as an exponent. E.g. the following expression should produce the BigFloat number BigFloat(-123, 997):

```
-0.123E+1000: BigFloat
```

import scala.util.FromDigits

The companion object of BigFloat defines an apply constructor method to construct a BigFloat from a digits string. Here is a possible implementation:

```
object BigFloat:
```

```
def apply(digits: String): BigFloat =
  val (mantissaDigits, givenExponent) =
      digits.toUpperCase.split('E') match
         case Array(mantissaDigits, edigits) =>
            val expo =
               try FromDigits.intFromDigits(edigits)
               catch case ex: FromDigits.NumberTooLarge =>
                  throw FromDigits.NumberTooLarge(s"exponent too large: $edigits
            (mantissaDigits, expo)
         case Array(mantissaDigits) =>
            (mantissaDigits, 0)
  val (intPart, exponent) =
      mantissaDigits.split('.') match
         case Array(intPart, decimalPart) =>
            (intPart ++ decimalPart, givenExponent - decimalPart.length)
         case Array(intPart) =>
```

```
(intPart, givenExponent)
BigFloat(BigInt(intPart), exponent)
```

To accept BigFloat literals, all that's needed in addition is a given instance of type FromDigits.Floating[BigFloat]:

```
given FromDigits: FromDigits.Floating[BigFloat] with
    def fromDigits(digits: String) = apply(digits)
end BigFloat
```

Note that the apply method does not check the format of the digits argument. It is assumed that only valid arguments are passed. For calls coming from the compiler that assumption is valid, since the compiler will first check whether a numeric literal has the correct format before it gets passed on to a conversion method.

7.1.0.5 Compile-Time Errors With the setup of the previous section, a literal like

```
1e10_0000_000_000: BigFloat
would be expanded by the compiler to
BigFloat.FromDigits.fromDigits("1e10000000000")
```

Evaluating this expression throws a NumberTooLarge exception at run time. We would like it to produce a compile-time error instead. We can achieve this by tweaking the BigFloat class with a small dose of metaprogramming. The idea is to turn the fromDigits method into a macro, i.e. make it an inline method with a splice as right hand side. To do this, replace the FromDigits instance in the BigFloat object by the following two definitions:

```
object BigFloat:
    ...

class FromDigits extends FromDigits.Floating[BigFloat]:
    def fromDigits(digits: String) = apply(digits)

given FromDigits with
    override inline def fromDigits(digits: String) = ${
    fromDigitsImpl('digits)
}
```

Note that an inline method cannot directly fill in for an abstract method, since it produces no code that can be executed at runtime. That is why we define an intermediary class FromDigits that contains a fallback implementation which is then overridden by the inline method in the FromDigits given instance. That method is defined in terms of a macro implementation method fromDigitsImpl. Here is its definition:

```
private def fromDigitsImpl(digits: Expr[String])(using ctx: Quotes): Expr[BigFloodigits.value match
```

```
case Some(ds) =>
    try
    val BigFloat(m, e) = apply(ds)
    '{BigFloat(${Expr(m)}, ${Expr(e)})}
    catch case ex: FromDigits.FromDigitsException =>
        ctx.error(ex.getMessage)
        '{BigFloat(0, 0)}
    case None =>
        '{apply($digits)}
end BigFloat
```

The macro implementation takes an argument of type Expr[String] and yields a result of type Expr[BigFloat]. It tests whether its argument is a constant string. If that is the case, it converts the string using the apply method and lifts the resulting BigFloat back to Expr level. For non-constant strings fromDigitsImpl(digits) is simply apply(digits), i.e. everything is evaluated at runtime in this case.

The interesting part is the catch part of the case where digits is constant. If the apply method throws a FromDigitsException, the exception's message is issued as a compile time error in the ctx.error(ex.getMessage) call.

With this new implementation, a definition like

7.2 Programmatic Structural Types

7.2.1 Motivation

Some usecases, such as modelling database access, are more awkward in statically typed languages than in dynamically typed languages: With dynamically typed languages, it's quite natural to model a row as a record or object, and to select entries with simple dot notation (e.g. row.columnName).

Achieving the same experience in statically typed language requires defining a class for every possible row arising from database manipulation (including rows arising from joins and projections) and setting up a scheme to map between a row and the class representing it.

This requires a large amount of boilerplate, which leads developers to trade the advantages of static typing for simpler schemes where colum names are represented as strings and passed to other operators (e.g. row.select("columnName")). This approach forgoes the advantages of static typing, and is still not as natural as the dynamically typed version.

Structural types help in situations where we would like to support simple dot notation in dynamic contexts without losing the advantages of static typing. They allow developers to use dot notation and configure how fields and methods should be resolved.

7.2.2 Example

Here's an example of a structural type Person:

```
class Record(elems: (String, Any)*) extends Selectable:
   private val fields = elems.toMap
   def selectDynamic(name: String): Any = fields(name)

type Person = Record { val name: String; val age: Int }
```

The type Person adds a *refinement* to its parent type Record that defines the two fields name and age. We say the refinement is *structural* since name and age are not defined in the parent type. But they exist nevertheless as members of class Person. For instance, the following program would print "Emma is 42 years old.":

```
val person = Record("name" -> "Emma", "age" -> 42).asInstanceOf[Person]
println(s"${person.name} is ${person.age} years old.")
```

The parent type Record in this example is a generic class that can represent arbitrary records in its elems argument. This argument is a sequence of pairs of labels of type String and values of type Any. When we create a Person as a Record we have to assert with a typecast that the record defines the right fields of the right types. Record itself is too weakly typed so the compiler cannot know this without help from the user. In practice, the connection between a structural type and its underlying generic representation would most likely be done by a database layer, and therefore would not be a concern of the end user.

Record extends the marker trait scala. Selectable and defines a method selectDynamic, which maps a field name to its value. Selecting a structural type member is done by calling this method. The person.name and person.age selections are translated by the Scala compiler to:

```
person.selectDynamic("name").asInstanceOf[String]
person.selectDynamic("age").asInstanceOf[Int]
```

Besides selectDynamic, a Selectable class sometimes also defines a method applyDynamic. This can then be used to translate function calls of structural members. So, if a is an instance of Selectable, a structural call like a.f(b, c) would translate to

```
a.applyDynamic("f")(b, c)
```

7.2.3 Using Java Reflection

Structural types can also be accessed using Java reflection. Example:

```
type Closeable = { def close(): Unit }

class FileInputStream:
   def close(): Unit

class Channel:
   def close(): Unit
```

Here, we define a structural type Closeable that defines a close method. There are various classes that have close methods, we just list FileInputStream and Channel as two examples. It would be easiest if the two classes shared a common interface that factors out the close method. But such factorings are often not possible if different libraries are combined in one application. Yet, we can still have methods that work on all classes with a close method by using the Closeable type. For instance,

```
import scala.reflect.Selectable.reflectiveSelectable

def autoClose(f: Closeable)(op: Closeable => Unit): Unit =
    try op(f) finally f.close()
```

The call f.close() has to use Java reflection to identify and call the close method in the receiver f. This needs to be enabled by an import of reflectiveSelectable shown above. What happens "under the hood" is then the following:

- The import makes available an implicit conversion that turns any type into a Selectable. f is wrapped in this conversion.
- The compiler then transforms the close call on the wrapped f to an applyDynamic call. The end result is:

```
reflectiveSelectable(f).applyDynamic("close")()
```

• The implementation of applyDynamic in reflectiveSelectable's result uses Java reflection to find and call a method close with zero parameters in the value referenced by f at runtime.

Structural calls like this tend to be much slower than normal method calls. The mandatory import of reflectiveSelectable serves as a signpost that something inefficient is going on.

Note: In Scala 2, Java reflection is the only mechanism available for structural types and it is automatically enabled without needing the reflectiveSelectable conversion. However, to warn against inefficient dispatch, Scala 2 requires a language import import scala.language.reflectiveCalls.

Before resorting to structural calls with Java reflection one should consider alternatives. For instance, sometimes a more a modular *and* efficient architecture can be obtained using type classes.

7.2.4 Extensibility

New instances of Selectable can be defined to support means of access other than Java reflection, which would enable usages such as the database access example given at the beginning of this document.

7.2.5 Local Selectable Instances

Local and anonymous classes that extend Selectable get more refined types than other classes. Here is an example:

```
trait Vehicle extends reflect.Selectable:
   val wheels: Int

val i3 = new Vehicle: // i3: Vehicle { val range: Int }
   val wheels = 4
   val range = 240

i3.range
```

The type of i3 in this example is Vehicle { val range: Int }. Hence, i3.range is well-formed. Since the base class Vehicle does not define a range field or method, we need structural dispatch to access the range field of the anonymous class that initializes id3. Structural dispatch is implemented by the base trait reflect. Selectable of Vehicle, which defines the necessary selectDynamic member.

Vehicle could also extend some other subclass of scala. Selectable that implements selectDynamic and applyDynamic differently. But if it does not extend a Selectable at all, the code would no longer typecheck:

```
trait Vehicle:
   val wheels: Int

val i3 = new Vehicle: // i3: Vehicle
   val wheels = 4
   val range = 240

i3.range // error: range is not a member of `Vehicle`
```

The difference is that the type of an anonymous class that does not extend Selectable is just formed from the parent type(s) of the class, without adding any refinements. Hence, i3 now has just type Vehicle and the selection i3.range gives a "member not found" error.

Note that in Scala 2 all local and anonymous classes could produce values with refined types. But members defined by such refinements could be selected only with the language import reflectiveCalls.

7.2.6 Relation with scala. Dynamic

There are clearly some connections with scala.Dynamic here, since both select members programmatically. But there are also some differences.

- Fully dynamic selection is not typesafe, but structural selection is, as long as the correspondence of the structural type with the underlying value is as stated.
- Dynamic is just a marker trait, which gives more leeway where and how to define reflective access operations. By contrast Selectable is a trait which declares the access operations.
- Two access operations, selectDynamic and applyDynamic are shared between both approaches. In Selectable, applyDynamic also may also take java.lang.Class arguments indicating the method's formal parameter types. Dynamic comes with updateDynamic.

More details

7.3 Rules for Operators

The rules for infix operators have changed in some parts:

First, an alphanumeric method can be used as an infix operator only if its definition carries an infix modifier. Second, it is recommended (but not enforced) to augment definitions of symbolic operators with <code>@targetName</code> annotations. Finally, a syntax change allows infix operators to be written on the left in a multi-line expression.

7.3.1 The infix Modifier

An infix modifier on a method definition allows using the method as an infix operation. Example:

```
import scala.annotation.targetName

trait MultiSet[T]:
   infix def union(other: MultiSet[T]): MultiSet[T]
   def difference(other: MultiSet[T]): MultiSet[T]

   @targetName("intersection")
   def *(other: MultiSet[T]): MultiSet[T]

end MultiSet

val s1, s2: MultiSet[Int]

s1 union s2  // OK
```

```
// also OK but unusual
s1 `union` s2
s1.union(s2)
                    // also OK
s1.difference(s2)
                    // OK
s1 `difference` s2
                   // OK
s1 difference s2
                    // gives a deprecation warning
                    // OK
s1 * s2
s1 `*` s2
                    // also OK, but unusual
s1.*(s2)
                    // also OK, but unusual
```

Infix operations involving alphanumeric operators are deprecated, unless one of the following conditions holds:

- the operator definition carries an infix modifier, or
- the operator was compiled with Scala 2, or
- the operator is followed by an opening brace.

An alphanumeric operator is an operator consisting entirely of letters, digits, the \$ and _characters, or any Unicode character c for which java.lang.Character.isIdentifierPart(c) returns true.

Infix operations involving symbolic operators are always allowed, so infix is redundant for methods with symbolic names.

The infix modifier can also be given to a type:

```
infix type or[X, Y]
val x: String or Int = ...
```

7.3.1.1 Motivation The purpose of the infix modifier is to achieve consistency across a code base in how a method or type is applied. The idea is that the author of a method decides whether that method should be applied as an infix operator or in a regular application. Use sites then implement that decision consistently.

7.3.1.2 Details

- 1. **infix** is a soft modifier. It is treated as a normal identifier except when in modifier position.
- 2. If a method overrides another, their infix annotations must agree. Either both are annotated with infix, or none of them are.
- 3. infix modifiers can be given to method definitions. The first non-receiver parameter list of an infix method must define exactly one parameter. Examples:

```
infix def op1(x: S): R // ok infix def op2[T](x: T)(y: S): R // ok infix def op3[T](x: T, y: S): R // error: two parameters
```

```
extension (x: A)
infix def op4(y: B): R  // ok
infix def op5(y1: B, y2: B): R  // error: two parameters
```

4. infix modifiers can also be given to type, trait or class definitions that have exactly two type parameters. An infix type like

```
infix type op[X, Y]
can be applied using infix syntax, i.e. A op B.
```

5. To smooth migration to Scala 3.0, alphanumeric operators will only be deprecated from Scala 3.1 onwards, or if the -source 3.1 option is given in Dotty/Scala 3.

7.3.2 The @targetName Annotation

It is recommended that definitions of symbolic operators carry a <code>@targetName</code> annotation that provides an encoding of the operator with an alphanumeric name. This has several benefits:

- It helps interoperability between Scala and other languages. One can call a Scala-defined symbolic operator from another language using its target name, which avoids having to remember the low-level encoding of the symbolic name.
- It helps legibility of stacktraces and other runtime diagnostics, where the userdefined alphanumeric name will be shown instead of the low-level encoding.
- It serves as a documentation tool by providing an alternative regular name as an alias of a symbolic operator. This makes the definition also easier to find in a search.

7.3.3 Syntax Change

Infix operators can now appear at the start of lines in a multi-line expression. Examples:

Previously, those expressions would have been rejected, since the compiler's semicolon inference would have treated the continuations ++ " world" or || xs.isEmpty as separate statements.

To make this syntax work, the rules are modified to not infer semicolons in front of leading infix operators. A *leading infix operator* is - a symbolic identifier such as +, or approx_==, or an identifier in backticks, - that starts a new line, - that precedes

a token on the same line that can start an expression, - and that is immediately followed by at least one space character ' '.

Example:

```
freezing | boiling
```

This is recognized as a single infix operation. Compare with:

```
freezing!boiling
```

This is seen as two statements, freezing and !boiling. The difference is that only the operator in the first example is followed by a space.

Another example:

```
println("hello")
???
??? match { case 0 => 1 }
```

This code is recognized as three different statements. ??? is syntactically a symbolic identifier, but neither of its occurrences is followed by a space and a token that can start an expression.

7.4 Wildcard Arguments in Types

The syntax of wildcard arguments in types has changed from _ to ?. Example:

```
List[?]
Map[? <: AnyRef, ? >: Null]
```

7.4.0.1 Motivation We would like to use the underscore syntax $_$ to stand for an anonymous type parameter, aligning it with its meaning in value parameter lists. So, just as $f(_)$ is a shorthand for the lambda $x \Rightarrow f(x)$, in the future $C[_]$ will be a shorthand for the type lambda $[X] \Rightarrow C[X]$. This makes higher-kinded types easier to use. It also removes the wart that, used as a type parameter, $F[_]$ means F is a type constructor whereas used as a type, $F[_]$ means it is a wildcard (i.e. existential) type. In the future, $F[_]$ will mean the same thing, no matter where it is used.

We pick? as a replacement syntax for wildcard types, since it aligns with Java's syntax.

7.4.0.2 Migration Strategy The migration to the new scheme is complicated, in particular since the kind projector compiler plugin still uses the reverse convention, with? meaning parameter placeholder instead of wildcard. Fortunately, kind projector has added * as an alternative syntax for?.

A step-by-step migration is made possible with the following measures:

- 1. In Scala 3.0, both _ and ? are legal names for wildcards.
- 2. In Scala 3.1, _ is deprecated in favor of ? as a name for a wildcard. A -rewrite option is available to rewrite one to the other.
- 3. In Scala 3.2, the meaning of _ changes from wildcard to placeholder for type parameter.
- 4. The Scala 3.1 behavior is already available today under the -source 3.1 setting.

To smooth the transition for codebases that use kind-projector, we adopt the following measures under the command line option -Ykind-projector:

- 1. In Scala 3.0, * is available as a type parameter placeholder.
- 2. In Scala 3.2, * is deprecated in favor of _. A -rewrite option is available to rewrite one to the other.
- 3. In Scala 3.3, * is removed again, and all type parameter placeholders will be expressed with _.

These rules make it possible to cross build between Scala 2 using the kind projector plugin and Scala 3.0 - 3.2 using the compiler option -Ykind-projector.

7.5 Changes in Type Checking *** TO BE FILLED IN ***

7.6 Changes in Type Inference

For more information, see the two presentations

- Scala 3, Type inference and You! by Guillaume Martres (September 2019)
- GADTs in Dotty by Aleksander Boruch-Gruszecki (July 2019).

7.7 Changes in Implicit Resolution

This section describes changes to the implicit resolution that apply both to the new givens and to the old-style implicits in Scala 3. Implicit resolution uses a new algorithm which caches implicit results more aggressively for performance. There are also some changes that affect implicits on the language level.

1. Types of implicit values and result types of implicit methods must be explicitly declared. Excepted are only values in local blocks where the type may still be inferred:

```
val y = {
  implicit val ctx = this.ctx // ok
   ...
}
```

2. Nesting is now taken into account for selecting an implicit. Consider for instance the following scenario:

```
def f(implicit i: C) = {
   def g(implicit j: C) = {
     implicitly[C]
   }
}
```

This will now resolve the **implicitly** call to j, because j is nested more deeply than i. Previously, this would have resulted in an ambiguity error. The previous possibility of an implicit search failure due to *shadowing* (where an implicit is hidden by a nested definition) no longer applies.

3. Package prefixes no longer contribute to the implicit search scope of a type. Example:

```
package p
given a: A = A()

object o:
    given b: B = B()
    type C
```

Both a and b are visible as implicits at the point of the definition of type C. However, a reference to p.o.C outside of package p will have only b in its implicit search scope but not a.

In more detail, here are the rules for what constitutes the implicit scope of a type:

Definition: A reference is an *anchor* if it refers to an object, a class, a trait, an abstract type, an opaque type alias, or a match type alias. References to packages and package objects are anchors only under -source:3.0-migration.

Definition: The *anchors* of a type T is a set of references defined as follows:

- 1. If T is a reference to an anchor, T itself plus, if T is of the form P#A, the anchors of P.
- 2. If T is an alias of U, the anchors of U.
- 3. If T is a reference to a type parameter, the union of the anchors of both of its bounds.
- 4. If T is a singleton reference, the anchors of its underlying type, plus, if T is of the form (P#x).type, the anchors of P.

- 5. If T is the this-type o.this of a static object o, the anchors of a term reference o.type to that object.
- 6. If T is some other type, the union of the anchors of each constituent type of T.

Definition: The *implicit scope* of a type T is the smallest set S of term references such that

- 1. If T is a reference to a class, S includes a reference to the companion object of the class, if it exists, as well as the implicit scopes of all of T's parent classes.
- 2. If T is a reference to an object, S includes T itself as well as the implicit scopes of all of T's parent classes.
- 3. If T is a reference to an opaque type alias named A, S includes a reference to an object A defined in the same scope as the type, if it exists, as well as the implicit scope of T's underlying type or bounds.
- 4. If T is a reference to an an abstract type or match type alias named A, S includes a reference to an object A defined in the same scope as the type, if it exists, as well as the implicit scopes of T's given bounds.
- 5. If T is a reference to an anchor of the form p.A then S also includes all term references on the path p.
- 6. If T is some other type, S includes the implicit scopes of all anchors of T.
- **4.** The treatment of ambiguity errors has changed. If an ambiguity is encountered in some recursive step of an implicit search, the ambiguity is propagated to the caller.

Example: Say you have the following definitions:

```
class A
class B extends C
class C
implicit def a1: A
implicit def a2: A
implicit def b(implicit a: A): B
implicit def c: C
```

and the query implicitly [C].

This query would now be classified as ambiguous. This makes sense, after all there are two possible solutions, b(a1) and b(a2), neither of which is better than the other and both of which are better than the third solution, c. By contrast, Scala 2 would have rejected the search for A as ambiguous, and subsequently have classified the query b(implicitly[A]) as a normal fail, which means that the alternative c would be chosen as solution!

Scala 2's somewhat puzzling behavior with respect to ambiguity has been exploited to implement the analogue of a "negated" search in implicit resolution, where a query Q1 fails if some other query Q2 succeeds and Q1 succeeds if Q2 fails. With the new cleaned up behavior these techniques no longer work. But there is now a new special type scala.util.Not which implements negation directly. For any query type Q: Not[Q] succeeds if and only if the implicit search for Q fails.

- 5. The treatment of divergence errors has also changed. A divergent implicit is treated as a normal failure, after which alternatives are still tried. This also makes sense: Encountering a divergent implicit means that we assume that no finite solution can be found on the corresponding path, but another path can still be tried. By contrast, most (but not all) divergence errors in Scala 2 would terminate the implicit search as a whole.
- **6.** Scala 2 gives a lower level of priority to implicit conversions with call-by-name parameters relative to implicit conversions with call-by-value parameters. Scala 3 drops this distinction. So the following code snippet would be ambiguous in Scala 3:

```
implicit def conv1(x: Int): A = new A(x)
implicit def conv2(x: => Int): A = new A(x)
def buzz(y: A) = ???
buzz(1) // error: ambiguous
```

7. The rule for picking a most specific alternative among a set of overloaded or implicit alternatives is refined to take context parameters into account. All else being equal, an alternative that takes some context parameters is taken to be less specific than an alternative that takes none. If both alternatives take context parameters, we try to choose between them as if they were methods with regular parameters. The following paragraph in the SLS §6.26.3 is affected by this change:

Original version:

An alternative A is *more specific* than an alternative B if the relative weight of A over B is greater than the relative weight of B over A.

Modified version:

An alternative A is more specific than an alternative B if

- the relative weight of A over B is greater than the relative weight of B over A, or
- the relative weights are the same, and A takes no implicit parameters but B does, or
- the relative weights are the same, both A and B take implicit parameters, and A is more specific than B if all implicit parameters in either alternative are replaced by regular parameters.
- 8. The previous disambiguation of implicits based on inheritance depth is refined to make it transitive. Transitivity is important to guarantee that search outcomes are compilation-order independent. Here's a scenario where the previous rules violated transitivity:

```
class A extends B
object A { given a ... }
class B
object B extends C { given b ... }
class C { given c }
```

Here a is more specific than b since the companion class A is a subclass of the companion class B. Also, b is more specific than c since object B extends class C. But a is not more specific than c. This means if a, b, c are all applicable implicits, it makes a difference in what order they are compared. If we compare b and c first, we keep b and drop c. Then, comparing a with b we keep a. But if we compare a with c first, we fail with an ambiguity error.

The new rules are as follows: An implicit a defined in A is more specific than an implicit b defined in B if

- A extends B, or
- A is an object and the companion class of A extends B, or
- A and B are objects, B does not inherit any implicit members from base classes (*), and the companion class of A extends the companion class of B.

Condition (*) is new. It is necessary to ensure that the defined relation is transitive.

7.8 Implicit Conversions

An *implicit conversion*, also called *view*, is a conversion that is applied by the compiler in several situations:

- 1. When an expression **e** of type T is encountered, but the compiler needs an expression of type S.
- 2. When an expression ${\tt e.m}$ where ${\tt e}$ has type T but T defines no member ${\tt m}$ is encountered.

In those cases, the compiler looks in the implicit scope for a conversion that can convert an expression of type T to an expression of type S (or to a type that defines a member m in the second case).

This conversion can be either:

- 1. An implicit def of type $T \Rightarrow S$ or $(\Rightarrow T) \Rightarrow S$
- 2. An implicit value of type scala.Conversion[T, S]

Defining an implicit conversion will emit a warning unless the import scala.language.implicitConversions is in scope, or the flag -language:implicitConversions is given to the compiler.

7.8.1 Examples

The first example is taken from scala. Predef. Thanks to this implicit conversion, it is possible to pass a scala. Int to a Java method that expects a java.lang. Integer

```
import scala.language.implicitConversions
implicit def int2Integer(x: Int): java.lang.Integer =
    x.asInstanceOf[java.lang.Integer]
```

The second example shows how to use Conversion to define an Ordering for an arbitrary type, given existing Orderings for other types:

More details

7.9 Changes in Overload Resolution

Overload resolution in Scala 3 improves on Scala 2 in two ways. First, it takes all argument lists into account instead of just the first argument list. Second, it can infer parameter types of function values even if they are in the first argument list.

7.9.1 Looking Beyond the First Argument List

Overloading resolution now can take argument lists into account when choosing among a set of overloaded alternatives. For example, the following code compiles in Scala 3, while it results in an ambiguous overload error in Scala 2:

To make this work, the rules for overloading resolution in SLS §6.26.3 are augmented as follows:

In a situation where a function is applied to more than one argument list, if overloading resolution yields several competing alternatives when $n \ge 1$ parameter lists are taken into account, then resolution re-tried using n + 1 argument lists.

This change is motivated by the new language feature extension methods, where emerges the need to do overload resolution based on additional argument blocks.

7.9.2 Parameter Types of Function Values

The handling of function values with missing parameter types has been improved. We can now pass such values in the first argument list of an overloaded application, provided that the remaining parameters suffice for picking a variant of the overloaded function. For example, the following code compiles in Scala 3, while it results in an missing parameter type error in Scala2:

```
def f(x: Int, f2: Int \Rightarrow Int) = f2(x)
def f(x: String, f2: String \Rightarrow String) = f2(x)
f("a", _.toUpperCase)
f(2, _ * 2)
```

To make this work, the rules for overloading resolution in SLS §6.26.3 are modified as follows:

Replace the sentence

Otherwise, let S1,...,Sm be the vector of types obtained by typing each argument with an undefined expected type.

with the following paragraph:

Otherwise, let S1,...,Sm be the vector of known types of all argument types, where the *known type* of an argument E is determined as followed:

- If E is a function value (p_1, ..., p_n) => B that misses some parameter types, the known type of E is (S_1, ..., S_n) => ?, where each S_i is the type of parameter p_i if it is given, or ? otherwise. Here ? stands for a wildcard type that is compatible with every other type.
- Otherwise the known type of E is the result of typing E with an undefined expected type.

A pattern matching closure

```
{ case P1 => B1 ... case P_n => B_n }
is treated as if it was expanded to the function value
x => x match { case P1 => B1 ... case P_n => B_n }
and is therefore also approximated with a ? => ? type.
```

7.10 Match Expressions

The syntactical precedence of match expressions has been changed. match is still a keyword, but it is used like an alphabetical operator. This has several consequences:

1. match expressions can be chained:

```
xs match {
   case Nil => "empty"
   case x :: xs1 => "nonempty"
} match {
   case "empty" => 0
   case "nonempty" => 1
}

(or, dropping the optional braces)

xs match
   case Nil => "empty"
   case x :: xs1 => "nonempty"

match
   case "empty" => 0
   case "nonempty" => 1
```

2. match may follow a period:

```
if xs.match
   case Nil => false
   case _ => true
then "nonempty"
else "empty"
```

3. The scrutinee of a match expression must be an InfixExpr. Previously the scrutinee could be followed by a type ascription: T, but this is no longer supported. So x: T match { ... } now has to be written (x: T) match { ... }.

7.10.1 Syntax

The new syntax of match expressions is as follows.

7.11 Vararg Patterns

The syntax of vararg patterns has changed. In the new syntax one writes varargs in patterns exactly like one writes them in expressions, using a : _* type annotation:

The old syntax, which is shorter but less regular, is no longer supported.

7.11.1 Compatibility considerations

To enable smooth cross compilation between Scala 2 and Scala 3, the compiler will accept both the old and the new syntax. Under the -source 3.1 setting, an error will be emitted when the old syntax is encountered. They will be enabled by default in version 3.1 of the language.

7.12 Pattern Bindings

In Scala 2, pattern bindings in val definitions and for expressions are loosely typed. Potentially failing matches are still accepted at compile-time, but may influence the program's runtime behavior. From Scala 3.1 on, type checking rules will be tightened so that warnings are reported at compile-time instead.

7.12.1 Bindings in Pattern Definitions

This code gives a compile-time warning in Scala 3.1 (and also in Scala 3.0 under the -source 3.1 setting) whereas it will fail at runtime with a ClassCastException in Scala 2. In Scala 3.1, a pattern binding is only allowed if the pattern is *irrefutable*, that is, if the right-hand side's type conforms to the pattern's type. For instance, the following is OK:

```
val pair = (1, true)
val (x, y) = pair
```

Sometimes one wants to decompose data anyway, even though the pattern is refutable. For instance, if at some point one knows that a list elems is non-empty one might want to decompose it like this:

```
val first :: rest = elems // error
```

This works in Scala 2. In fact it is a typical use case for Scala 2's rules. But in Scala 3.1 it will give a warning. One can avoid the warning by marking the right-hand side with an @unchecked annotation:

```
val first :: rest = elems: @unchecked // OK
```

This will make the compiler accept the pattern binding. It might give an error at runtime instead, if the underlying assumption that **elems** can never be empty is wrong.

7.12.2 Pattern Bindings in for Expressions

Analogous changes apply to patterns in for expressions. For instance:

```
val elems: List[Any] = List((1, 2), "hello", (3, 4))
for (x, y) <- elems yield (y, x) // error: pattern's type (Any, Any) is more specified (y, x) // than the right hand side expression's type Any</pre>
```

This code gives a compile-time warning in Scala 3.1 whereas in Scala 2 the list elems is filtered to retain only the elements of tuple type that match the pattern (x, y). The filtering functionality can be obtained in Scala 3 by prefixing the pattern with case:

```
for case (x, y) \leftarrow \text{elems yield } (y, x) // \text{returns } List((2, 1), (4, 3))
```

7.12.3 Syntax Changes

Generators in for expressions may be prefixed with case.

```
Generator ::= ['case'] Pattern1 '<-' Expr
```

7.12.4 Migration

The new syntax is supported in Scala 3.0. However, to enable smooth cross compilation between Scala 2 and Scala 3, the changed behavior and additional type checks are only enabled under the -source 3.1 setting. They will be enabled by default in version 3.1 of the language.

7.13 Option-less pattern matching

The implementation of pattern matching in Scala 3 was greatly simplified compared to Scala 2. From a user perspective, this means that Scala 3 generated patterns are a *lot* easier to debug, as variables all show up in debug modes and positions are correctly preserved.

Scala 3 supports a superset of Scala 2 extractors.

7.13.1 Extractors

Extractors are objects that expose a method unapply or unapplySeq:

```
def unapply[A](x: T)(implicit x: B): U
def unapplySeq[A](x: T)(implicit x: B): U
```

Extractors that expose the method unapply are called fixed-arity extractors, which work with patterns of fixed arity. Extractors that expose the method unapplySeq are called variadic extractors, which enables variadic patterns.

7.13.1.1 Fixed-Arity Extractors Fixed-arity extractors expose the following signature:

```
def unapply[A](x: T)(implicit x: B): U
```

The type U conforms to one of the following matches:

- Boolean match
- Product match

Or U conforms to the type R:

```
type R = {
  def isEmpty: Boolean
  def get: S
}
```

and S conforms to one of the following matches:

- single match
- name-based match

The former form of unapply has higher precedence, and *single match* has higher precedence over *name-based match*.

A usage of a fixed-arity extractor is irrefutable if one of the following condition holds:

- U = true
- the extractor is used as a product match
- U = Some [T] (for Scala 2 compatibility)
- U <: R and U <: { def isEmpty: false }

7.13.1.2 Variadic Extractors Variadic extractors expose the following signature:

```
def unapplySeq[A](x: T)(implicit x: B): U
```

The type U conforms to one of the following matches:

- sequence match
- product-sequence match

Or U conforms to the type R:

```
type R = {
  def isEmpty: Boolean
  def get: S
}
```

and S conforms to one of the two matches above.

The former form of unapplySeq has higher priority, and sequence match has higher precedence over product-sequence match.

A usage of a variadic extractor is irrefutable if one of the following conditions holds:

- the extractor is used directly as a sequence match or product-sequence match
- U = Some[T] (for Scala2 compatibility)
- U <: R and U <: { def isEmpty: false }

7.13.2 Boolean Match

- U =:= Boolean
- Pattern-matching on exactly 0 pattern

For example:

```
object Even:
    def unapply(s: String): Boolean = s.size % 2 == 0

"even" match
    case s @ Even() => println(s"$s has an even number of characters")
    case s => println(s"$s has an odd number of characters")

// even has an even number of characters
```

7.13.3 Product Match

- U <: Product
- N > 0 is the maximum number of consecutive (parameterless def or val) _1: P1 ... N: PN members in U
- Pattern-matching on exactly N patterns with types P1, P2, ..., PN

For example:

```
class FirstChars(s: String) extends Product:
    def _1 = s.charAt(0)
    def _2 = s.charAt(1)

// Not used by pattern matching: Product is only used as a marker trait.
    def canEqual(that: Any): Boolean = ???
    def productArity: Int = ???
    def productElement(n: Int): Any = ???
object FirstChars:
```

```
def unapply(s: String): FirstChars = new FirstChars(s)
"Hi!" match
   case FirstChars(char1, char2) =>
      println(s"First: $char1; Second: $char2")
// First: H: Second: i
7.13.4 Single Match
  • If there is exactly 1 pattern, pattern-matching on 1 pattern with type U
class Nat(val x: Int):
   def get: Int = x
   def isEmpty = x < 0
object Nat:
   def unapply(x: Int): Nat = new Nat(x)
5 match
   case Nat(n) => println(s"$n is a natural number")
   case _
             => ()
// 5 is a natural number
7.13.5 Name-based Match
  • N > 1 is the maximum number of consecutive (parameterless def or val) _1:
    P1 ... N: PN members in U
  • Pattern-matching on exactly N patterns with types P1, P2, ..., PN
object ProdEmpty:
   def 1: Int = ???
   def _2: String = ???
   def isEmpty = true
   def unapply(s: String): this.type = this
   def get = this
"" match
   case ProdEmpty(_, _) => ???
   case _ => ()
7.13.6 Sequence Match
  • U <: X, T2 and T3 conform to T1
type X = {
   def lengthCompare(len: Int): Int // or, `def length: Int`
   def apply(i: Int): T1
```

```
def drop(n: Int): scala.Seq[T2]
  def toSeq: scala.Seq[T3]
}
```

- Pattern-matching on *exactly* N simple patterns with types T1, T1, ..., T1, where N is the runtime size of the sequence, or
- Pattern-matching on >= N simple patterns and a varary pattern (e.g., xs: _*) with types T1, T1, ..., T1, Seq[T1], where N is the minimum size of the sequence.

```
object CharList:
```

```
def unapplySeq(s: String): Option[Seq[Char]] = Some(s.toList)

"example" match
  case CharList(c1, c2, c3, c4, _, _, _) =>
      println(s"$c1,$c2,$c3,$c4")
  case _ =>
      println("Expected *exactly* 7 characters!")

// e,x,a,m
```

7.13.7 Product-Sequence Match

- U <: Product
- N > 0 is the maximum number of consecutive (parameterless def or val) _1: P1 ... _N: PN members in U
- PN conforms to the signature X defined in Seq Pattern
- Pattern-matching on exactly >= N patterns, the first N 1 patterns have types P1, P2, ... P(N-1), the type of the remaining patterns are determined as in Seq Pattern.

```
class Foo(val name: String, val children: Int *)
object Foo:
    def unapplySeq(f: Foo): Option[(String, Seq[Int])] =
        Some((f.name, f.children))

def foo(f: Foo) = f match
    case Foo(name, ns : _*) =>
    case Foo(name, x, y, ns : *) =>
```

There are plans for further simplification, in particular to factor out *product match* and *name-based match* into a single type of extractor.

7.13.8 Type testing

Abstract type testing with ClassTag is replaced with TypeTest or the alias Typeable.

• pattern _: X for an abstract type requires a TypeTest in scope

• pattern $x \in X()$ for an unapply that takes an abstract type requires a TypeTest in scope

More details on TypeTest

f2: Int => List[Int]

7.14 Automatic Eta Expansion 🗹

The conversion of *methods* into *functions* has been improved and happens automatically for methods with one or more parameters.

```
def m(x: Boolean, y: String)(z: Int): List[Int]
val f1 = m
val f2 = m(true, "abc")
This creates two function values:
f1: (Boolean, String) => Int => List[Int]
```

The syntax m is no longer needed and will be deprecated in the future.

7.14.1 Automatic eta-expansion and nullary methods

Automatic eta expansion does not apply to "nullary" methods that take an empty parameter list.

```
def next(): T
```

Given a simple reference to next does not auto-convert to a function. One has to write explicitly () => next() to achieve that Once again since the _ is going to be deprecated it's better to write it this way rather than next _.

The reason for excluding nullary methods from automatic eta expansion is that Scala implicitly inserts the () argument, which would conflict with eta expansion. Automatic () insertion is limited in Scala 3, but the fundamental ambiguity remains.

More details

7.15 Changes in Compiler Plugins

Compiler plugins are supported by Dotty (and Scala 3) since 0.9. There are two notable changes compared to scalac:

- No support for analyzer plugins
- Added support for research plugins

Analyzer plugins in scalac run during type checking and may influence normal type checking. This is a very powerful feature but for production usages, a predictable and consistent type checker is more important.

For experimentation and research, Scala 3 introduces research plugin. Research plugins are more powerful than scalac analyzer plugins as they let plugin authors customize the whole compiler pipeline. One can easily replace the standard typer by

a custom one or create a parser for a domain-specific language. However, research plugins are only enabled for nightly or snaphot releases of Scala 3.

Common plugins that add new phases to the compiler pipeline are called *standard plugins* in Scala 3. In terms of features, they are similar to **scalac** plugins, despite minor changes in the API.

7.15.1 Using Compiler Plugins

Both standard and research plugins can be used with scalac by adding the -Xplugin: option:

```
scalac -Xplugin:pluginA.jar -Xplugin:pluginB.jar Test.scala
```

The compiler will examine the jar provided, and look for a property file named plugin.properties in the root directory of the jar. The property file specifies the fully qualified plugin class name. The format of a property file is as follows:

```
pluginClass=dividezero.DivideZero
```

This is different from scalar plugins that required a scalar-plugin.xml file.

Starting from 1.1.5, sbt also supports Scala 3 compiler plugins. Please refer to the sbt documentation for more information.

7.15.2 Writing a Standard Compiler Plugin

class DivideZeroPhase extends PluginPhase:

Here is the source code for a simple compiler plugin that reports integer divisions by zero as errors.

```
package dividezero
```

```
import dotty.tools.dotc.ast.Trees._
import dotty.tools.dotc.ast.tpd
import dotty.tools.dotc.core.Constants.Constant
import dotty.tools.dotc.core.Contexts.Context
import dotty.tools.dotc.core.Decorators._
import dotty.tools.dotc.core.StdNames._
import dotty.tools.dotc.core.Symbols._
import dotty.tools.dotc.plugins.{PluginPhase, StandardPlugin}
import dotty.tools.dotc.transform.{Pickler, Staging}

class DivideZero extends StandardPlugin:
   val name: String = "divideZero"
   override val description: String = "divide zero check"

def init(options: List[String]): List[PluginPhase] =
        (new DivideZeroPhase) :: Nil
```

```
import tpd._
val phaseName = "divideZero"
override val runsAfter = Set(Pickler.name)
override val runsBefore = Set(Staging.name)
override def transformApply(tree: Apply)(implicit ctx: Context): Tree =
  tree match
      case Apply(Select(rcvr, nme.DIV), List(Literal(Constant(0))))
      if rcvr.tpe <:< defn.IntType =>
         report.error("dividing by zero", tree.pos)
      case _ =>
         ()
   tree
```

end DivideZeroPhase

The plugin main class (DivideZero) must extend the trait StandardPlugin and implement the method init that takes the plugin's options as argument and returns a list of PluginPhases to be inserted into the compilation pipeline.

Our plugin adds one compiler phase to the pipeline. A compiler phase must extend the PluginPhase trait. In order to specify when the phase is executed, we also need to specify a runsBefore and runsAfter constraints that are list of phase names.

We can now transform trees by overriding methods like transformXXX.

7.15.3 Writing a Research Compiler Plugin

Here is a template for research plugins.

```
import dotty.tools.dotc.core.Contexts.Context
import dotty.tools.dotc.core.Phases.Phase
import dotty.tools.dotc.plugins.ResearchPlugin
class DummyResearchPlugin extends ResearchPlugin:
  val name: String = "dummy"
  override val description: String = "dummy research plugin"
  def init(options: List[String], phases: List[List[Phase]])(implicit ctx: Context
     phases
end DummyResearchPlugin
```

A research plugin must extend the trait ResearchPlugin and implement the method init that takes the plugin's options as argument as well as the compiler pipeline in the form of a list of compiler phases. The method can replace, remove or add any phases to the pipeline and return the updated pipeline.

7.16 Lazy Vals Initialization 🗹

Scala 3 implements Version 6 of the SIP-20 improved lazy vals initialization proposal.

7.16.1 Motivation

The newly proposed lazy val initialization mechanism aims to eliminate the acquisition of resources during the execution of the lazy val initializer block, thus reducing the possibility of a deadlock. The concrete deadlock scenarios that the new lazy val initialization scheme eliminates are summarized in the SIP-20 document.

7.16.2 Implementation

```
Given a lazy field of the form:
class Foo {
  lazy val bar = <RHS>
The Scala 3 compiler will generate code equivalent to:
class Foo {
  import scala.runtime.LazyVals
  var value 0: Int =
  var bitmap: Long = OL
  val bitmap offset: Long = LazyVals.getOffset(classOf[LazyCell], "bitmap")
  def bar(): Int = {
    while (true) {
      val flag = LazyVals.get(this, bitmap_offset)
      val state = LazyVals.STATE(flag, <field-id>)
      if (state == <state-3>) {
        return value_0
      } else if (state == <state-0>) {
        if (LazyVals.CAS(this, bitmap_offset, flag, <state-1>, <field-id>)) {
          try {
            val result = <RHS>
            value 0 = result
            LazyVals.setFlag(this, bitmap_offset, <state-3>, <field-id>)
            return result
          }
          catch {
            case ex =>
              LazyVals.setFlag(this, bitmap_offset, <state-0>, <field-id>)
              throw ex
          }
        }
      } else /* if (state == <state-1> // state == <state-2>) */ {
```

```
LazyVals.wait4Notification(this, bitmap_offset, flag, <field-id>)
}
}
```

The state of the lazy val <state-i> is represented with 4 values: 0, 1, 2 and 3. The state 0 represents a non-initialized lazy val. The state 1 represents a lazy val that is currently being initialized by some thread. The state 2 denotes that there are concurrent readers of the lazy val. The state 3 represents a lazy val that has been initialized. <field-id> is the id of the lazy val. This id grows with the number of volatile lazy vals defined in the class.

7.16.3 Note on recursive lazy vals

Ideally recursive lazy vals should be flagged as an error. The current behavior for recursive lazy vals is undefined (initialization may result in a deadlock).

7.16.4 Reference

• SIP-20

7.17 Main Methods

Scala 3 offers a new way to define programs that can be invoked from the command line: A @main annotation on a method turns this method into an executable program. Example:

```
@main def happyBirthday(age: Int, name: String, others: String*) =
    val suffix =
        age % 100 match
        case 11 | 12 | 13 => "th"
        case _ =>
            age % 10 match
        case 1 => "st"
        case 2 => "nd"
        case 3 => "rd"
        case _ => "th"
    val bldr = new StringBuilder(s"Happy $age$suffix birthday, $name")
    for other <- others do bldr.append(" and ").append(other)
    bldr.toString</pre>
```

This would generate a main program happyBirthday that could be called like this

```
> scala happyBirthday 23 Lisa Peter
Happy 23rd Birthday, Lisa and Peter!
```

A @main annotated method can be written either at the top-level or in a statically accessible object. The name of the program is in each case the name of the

method, without any object prefixes. The <code>@main</code> method can have an arbitrary number of parameters. For each parameter type there must be an instance of the <code>scala.util.FromString</code> type class that is used to convert an argument string to the required parameter type. The parameter list of a main method can end in a repeated parameter that then takes all remaining arguments given on the command line.

The program implemented from a @main method checks that there are enough arguments on the command line to fill in all parameters, and that argument strings are convertible to the required types. If a check fails, the program is terminated with an error message.

Examples:

```
> scala happyBirthday 22
Illegal command line after first argument: more arguments expected
> scala happyBirthday sixty Fred
Illegal command line: java.lang.NumberFormatException: For input string: "sixty"
The Scala compiler generates a program from a @main method f as follows:
```

- It creates a class named f in the package where the @main method was found
- The class has a static method main with the usual signature. It takes an Array[String] as argument and returns Unit.
- The generated main method calls method f with arguments converted using methods in the scala.util.CommandLineParser object.

For instance, the happyBirthDay method above would generate additional code equivalent to the following class:

```
final class happyBirthday:
   import scala.util.{CommandLineParser => CLP}
   <static> def main(args: Array[String]): Unit =
        try
        happyBirthday(
            CLP.parseArgument[Int](args, 0),
            CLP.parseArgument[String](args, 1),
            CLP.parseRemainingArguments[String](args, 2))
   catch
      case error: CLP.ParseError => CLP.showError(error)
```

Note: The <static> modifier above expresses that the main method is generated as a static method of class happyBirthDay. It is not available for user programs in Scala. Regular "static" members are generated in Scala using objects instead.

Omain methods are the recommended scheme to generate programs that can be invoked from the command line in Scala 3. They replace the previous scheme to write program as objects with a special App parent class. In Scala 2, happyBirthday could be written also like this:

```
object happyBirthday extends App:
   // needs by-hand parsing of arguments vector
...
```

The previous functionality of App, which relied on the "magic" DelayedInit trait, is no longer available. App still exists in limited form for now, but it does not support command line arguments and will be deprecated in the future. If programs need to cross-build between Scala 2 and Scala 3, it is recommended to use an explicit main method with an Array[String] argument instead.

7.18 Escapes in interpolations

In Scala 2 there was no straightforward way to represent a single quote character " in a single quoted interpolation. A \ character can't be used for that because interpolators themselves decide how to handle escaping, so the parser doesn't know whether the " should be escaped or used as a terminator.

In Scala 3, you can use the \$ meta character of interpolations to escape a " character.

```
val inventor = "Thomas Edison"
val interpolation = s"as $inventor said: $"The three great essentials to achieve
```

8 Dropped Features

8.1 Dropped: DelayedInit

The special handling of the DelayedInit trait is no longer supported.

One consequence is that the App class, which used DelayedInit is now partially broken. You can still use App as a simple way to set up a main program. Example:

```
object HelloWorld extends App {
   println("Hello, world!")
}
```

However, the code is now run in the initializer of the object, which on some JVM's means that it will only be interpreted. So, better not use it for benchmarking! Also, if you want to access the command line arguments, you need to use an explicit main method for that.

```
object Hello:
   def main(args: Array[String]) =
        println(s"Hello, ${args(0)}")
```

On the other hand, Scala 3 offers a convenient alternative to such "program" objects with @main methods.

8.2 Dropped: Scala 2 Macros

The previous, experimental macro system has been dropped.

Instead, there is a cleaner, more restricted system based on two complementary concepts: inline and '{ ... }/\${ ... } code generation. '{ ... } delays the compilation of the code and produces an object containing the code, dually \${ ... } evaluates an expression which produces code and inserts it in the surrounding \${ ... }. In this setting, a definition marked as inlined containing a \${ ... } is a macro, the code inside the \${ ... } is executed at compile-time and produces code in the form of '{ ... }. Additionally, the contents of code can be inspected and created with a more complex reflection API (TASTy Reflect) as an extension of '{ ... }/\${ ... } framework.

- inline has been implemented in Scala 3.
- Quotes '{ ... } and splices \${ ... } has been implemented in Scala 3.
- TASTy reflect provides more complex tree based APIs to inspect or create quoted code.

8.3 Dropped: Existential types

Existential types using forSome (as in SLS §3.2.12) have been dropped. The reasons for dropping them are:

- Existential types violate a type soundness principle on which DOT and Scala 3 are constructed. That principle says that every prefix (p, respectively S) of a type selection p.T or S#T must either come from a value constructed at runtime or refer to a type that is known to have only good bounds.
- Existential types create many difficult feature interactions with other Scala constructs.
- Existential types largely overlap with path-dependent types, so the gain of having them is relatively minor.

Existential types that can be expressed using only wildcards (but not forSome) are still supported, but are treated as refined types. For instance, the type

```
Map[ <: AnyRef, Int]</pre>
```

is treated as the type Map, where the first type parameter is upper-bounded by AnyRef and the second type parameter is an alias of Int.

When reading class files compiled with Scala 2, Scala 3 will do a best effort to approximate existential types with its own types. It will issue a warning that a precise emulation is not possible.

8.4 Dropped: General Type Projection

Scala so far allowed general type projection T#A where T is an arbitrary type and A names a type member of T.

Scala 3 disallows this if T is an abstract type (class types and type aliases are fine). This change was made because unrestricted type projection is unsound.

This restriction rules out the type-level encoding of a combinator calculus.

To rewrite code using type projections on abstract types, consider using pathdependent types or implicit parameters.

8.5 Dropped: Do-While

The syntax construct

```
do <body> while <cond>
```

is no longer supported. Instead, it is recommended to use the equivalent while loop below:

```
while ({ <body> ; <cond> }) ()
For instance, instead of
```

```
do
    i += 1
while (f(i) == 0)
```

one writes

```
while
    i += 1
    f(i) == 0
do ()
```

The idea to use a block as the condition of a while also gives a solution to the "loop-and-a-half" problem. Here is another example:

```
while
   val x: Int = iterator.next
   x >= 0
do print(".")
```

8.5.0.1 Why Drop The Construct?

- do-while is used relatively rarely and it can expressed faithfully using just while. So there seems to be little point in having it as a separate syntax construct.
- Under the new syntax rules do is used as a statement continuation, which would clash with its meaning as a statement introduction.

8.6 Dropped: Procedure Syntax

Procedure syntax

```
def f() { ... }
```

has been dropped. You need to write one of the following instead:

```
def f() = { ... }
def f(): Unit = { ... }
```

Scala 3 accepts the old syntax under the -source:3.0-migration option. If the -migration option is set, it can even rewrite old syntax to new. The ScalaFix tool also can rewrite procedure syntax to make it Scala 3 compatible.

8.7 Dropped: Package Objects

Package objects

```
package object p {
  val a = ...
  def b = ...
}
```

will be dropped. They are still available in Scala 3.0, but will be deprecated and removed afterwards.

Package objects are no longer needed since all kinds of definitions can now be written at the top-level. Example:

```
package p
type Labelled[T] = (String, T)
val a: Labelled[Int] = ("count", 1)
def b = a._2

case class C()
extension (x: C) def pair(y: C) = (x, y)
```

There may be several source files in a package containing such top-level definitions, and source files can freely mix top-level value, method, and type definitions with classes and objects.

The compiler generates synthetic objects that wrap top-level definitions falling into one of the following categories:

- all pattern, value, method, and type definitions,
- implicit classes and objects,
- companion objects of opaque type aliases.

If a source file src.scala contains such top-level definitions, they will be put in a synthetic object named src\$package. The wrapping is transparent, however. The definitions in src can still be accessed as members of the enclosing package.

Note 1: This means that the name of a source file containing wrapped top-level definitions is relevant for binary compatibility. If the name changes, so does the name of the generated object and its class.

Note 2: A top-level main method def main(args: Array[String]): Unit = ... is wrapped as any other method. If it appears in a source file src.scala, it could be invoked from the command line using a command like scala src\$package. Since the "program name" is mangled it is recommended to always put main methods in explicitly named objects.

Note 3: The notion of **private** is independent of whether a definition is wrapped or not. A **private** top-level definition is always visible from everywhere in the enclosing package.

Note 4: If several top-level definitions are overloaded variants with the same name, they must all come from the same source file.

8.8 Dropped: Early Initializers

Early initializers of the form

```
class C extends { ... } with SuperClass ...
```

have been dropped. They were rarely used, and mostly to compensate for the lack of trait parameters, which are now directly supported in Scala 3.

8.9 Dropped: Class shadowing

Scala 2 so far allowed patterns like this:

```
class Base {
  class Ops { ... }
}
class Sub extends Base {
  class Ops { ... }
}
```

Scala 3 rejects this with the error message:

-- class definitions cannot be overridden

The issue is that the two Ops classes *look* like one overrides the other, but classes in Scala 2 cannot be overridden. To keep things clean (and its internal operations consistent) the Scala 3 compiler forces you to rename the inner classes so that their names are different.

More details

8.10 Dropped: Limit 22

The limits of 22 for the maximal number of parameters of function types and the maximal number of fields in tuple types have been dropped.

- Functions can now have an arbitrary number of parameters. Functions beyond Function22 are erased to a new trait scala.FunctionXXL
- Tuples can also have an arbitrary number of fields. Tuples beyond Tuple22 are erased to a new trait scala. TupleXXL. Furthermore, they support generic operation such as concatenation and indexing.

Both of these are implemented using arrays.

8.11 Dropped: XML literals

XML Literals are still supported, but will be dropped in the near future, to be replaced with XML string interpolation:

```
xml""" ... """
```

8.12 Dropped: Symbol literals 💆

Symbol literals are no longer supported.

The scala.Symbol class still exists, so a literal translation of the symbol literal 'xyz is Symbol("xyz"). However, it is recommended to use a plain string literal "xyz" instead. (The Symbol class will be deprecated and removed in the future).

8.13 Dropped: Auto-Application

Previously an empty argument list () was implicitly inserted when calling a nullary method without arguments. Example:

```
def next(): T = ...
next  // is expanded to next()
In Scala 3, this idiom is an error.
next
^
missing arguments for method next
```

In Scala 3, the application syntax has to follow exactly the parameter syntax. Excluded from this rule are methods that are defined in Java or that override methods defined in Java. The reason for being more lenient with such methods is that otherwise everyone would have to write

```
xs.toString().length()
instead of
xs.toString.length
```

The latter is idiomatic Scala because it conforms to the uniform access principle. This principle states that one should be able to change an object member from a field to a non-side-effecting method and back without affecting clients that access the member. Consequently, Scala encourages to define such "property" methods without a () parameter list whereas side-effecting methods should be defined with it. Methods defined in Java cannot make this distinction; for them a () is always mandatory. So Scala fixes the problem on the client side, by allowing the parameterless references. But where Scala allows that freedom for all method references, Scala 3 restricts it to references of external methods that are not defined themselves in Scala 3.

For reasons of backwards compatibility, Scala 3 for the moment also auto-inserts () for nullary methods that are defined in Scala 2, or that override a method defined in Scala 2. It turns out that, because the correspondence between definition and call was not enforced in Scala so far, there are quite a few method definitions in Scala 2 libraries that use () in an inconsistent way. For instance, we find in scala.math.Numeric

```
def toInt(): Int
```

whereas toInt is written without parameters everywhere else. Enforcing strict parameter correspondence for references to such methods would project the inconsistencies to client code, which is undesirable. So Scala 3 opts for more leniency when

type-checking references to such methods until most core libraries in Scala 2 have been cleaned up.

Stricter conformance rules also apply to overriding of nullary methods. It is no longer allowed to override a parameterless method by a nullary method or *vice versa*. Instead, both methods must agree exactly in their parameter lists.

```
class A:
    def next(): Int

class B extends A:
    def next: Int // overriding error: incompatible type
```

Methods overriding Java or Scala 2 methods are again exempted from this requirement.

8.13.0.1 Migrating code Existing Scala code with inconsistent parameters can still be compiled in Scala 3 under -source 3.0-migration. When paired with the -rewrite option, the code will be automatically rewritten to conform to Scala 3's stricter checking.

8.13.0.2 Reference For more information, see Issue #2570 and PR #2716.

8.14 Dropped: Weak conformance

In some situations, Scala used a *weak conformance* relation when testing type compatibility or computing the least upper bound of a set of types. The principal motivation behind weak conformance was to make an expression like this have type List[Double]:

```
List(1.0, math.sqrt(3.0), 0, -3.3) // : List[Double]
```

It's "obvious" that this should be a List[Double]. However, without some special provision, the least upper bound of the lists's element types (Double, Double, Int, Double) would be AnyVal, hence the list expression would be given type List[AnyVal].

A less obvious example is the following one, which was also typed as a List[Double], using the weak conformance relation.

```
val n: Int = 3
val c: Char = 'X'
val d: Double = math.sqrt(3.0)
List(n, c, d) // used to be: List[Double], now: List[AnyVal]
```

Here, it is less clear why the type should be widened to List[Double], a List[AnyVal] seems to be an equally valid – and more principled – choice.

Weak conformance applies to all "numeric" types (including Char), and independently of whether the expressions are literals or not. However, in hindsight, the only intended use case is for *integer literals* to be adapted to the type of the other

expressions. Other types of numerics have an explicit type annotation embedded in their syntax (f, d, ., L or ' for Chars) which ensures that their author really meant them to have that specific type).

Therefore, Scala 3 drops the general notion of weak conformance, and instead keeps one rule: Int literals are adapted to other numeric types if necessary.

More details

8.15 Deprecated: Nonlocal Returns

Returning from nested anonymous functions has been deprecated.

Nonlocal returns are implemented by throwing and catching scala.runtime.NonLocalReturnExcept s. This is rarely what is intended by the programmer. It can be problematic because of the hidden performance cost of throwing and catching exceptions. Furthermore, it is a leaky implementation: a catch-all exception handler can intercept a NonLocalReturnException.

A drop-in library replacement is provided in scala.util.control.NonLocalReturns. Example:

```
import scala.util.control.NonLocalReturns._
extension [T](xs: List[T])
  def has(elem: T): Boolean = returning {
     for x <- xs do
        if x == elem then throwReturn(true)
     false
  }

@main def test =
  val xs = List(1, 2, 3, 4, 5)
  assert(xs.has(2) == xs.contains(2))</pre>
```

8.16 Dropped: private[this] and protected[this]

The private[this] and protected[this] access modifiers are deprecated and will be phased out.

Previously, these modifiers were needed for

- avoiding the generation of getters and setters
- excluding code under a private[this] from variance checks. (Scala 2 also excludes protected[this] but this was found to be unsound and was therefore removed).

The compiler now infers for private members the fact that they are only accessed via this. Such members are treated as if they had been declared private[this]. protected[this] is dropped without a replacement.

9 Scala 3 Syntax Summary 💆

The following descriptions of Scala tokens uses literal characters 'c' when referring to the ASCII fragment $\u0000 - \u007F$.

Unicode escapes are used to represent the Unicode character with the given hexadecimal code:

```
UnicodeEscape ::= '\' 'u' {'u'} hexDigit hexDigit hexDigit
hexDigit ::= '0' | ... | '9' | 'A' | ... | 'F' | 'a' | ... | 'f'
```

Informal descriptions are typeset as "some comment".

9.0.0.1 Lexical Syntax The lexical syntax of Scala is given by the following grammar in EBNF form.

```
'\u0020' | '\u0009' | '\u000D' | '\u000A'
whiteSpace
                 : :=
                      'A' | ... | 'Z' | '\$' | ' ' "... and Unicode category Lu"
upper
                 ::=
lower
                 ::=
                      'a' | ... | 'z' "... and Unicode category Ll"
                      upper | lower "... and Unicode categories Lo, Lt, N1"
letter
                      '0' | ... | '9'
digit
                 ::=
                      ((, | ,), | ,(|, | ,1, | ,4, | ,4, | ,1, | ,1, | ,1, | ,1, |
paren
                 ::=
                      (`) | (|) | (|) | (;) | (;)
delim
                 : :=
                      "printableChar not matched by (whiteSpace | upper |
opchar
                       lower | letter | digit | paren | delim | opchar |
                       Unicode_Sm | Unicode_So)"
                      "all characters in [\u0020, \u007F] inclusive"
printableChar
                      '\' ('b' | 't' | 'n' | 'f' | 'r' | '"' | ''' | '\')
charEscapeSeq
                 ::=
                      opchar {opchar}
                 ::=
op
                 ::= lower idrest
varid
alphaid
                 ::= upper idrest
                      varid
plainid
                 ::=
                      alphaid
                   op
id
                 ::= plainid
                      '`' { charNoBackQuoteOrNewline | UnicodeEscape | charEscapeSe
                   idrest
                 ::= {letter | digit} ['_' op]
                      ''' alphaid
quoteId
                 ::=
                      (decimalNumeral | hexNumeral) ['L' | '1']
integerLiteral
                 ::=
                      '0' | nonZeroDigit [{digit | '_'} digit]
decimalNumeral
                 ::=
                      '0' ('x' | 'X') hexDigit [{hexDigit | ' '} hexDigit]
hexNumeral
                 : :=
                      '1' | ... | '9'
nonZeroDigit
                 ::=
floatingPointLiteral
```

[decimalNumeral] '.' digit [{digit | '_'} digit] [exponentPar

decimalNumeral exponentPart [floatType]

```
| decimalNumeral floatType
                     ('E' | 'e') ['+' | '-'] digit [{digit | ' '} digit]
exponentPart
                 ::=
                 ::= 'F' | 'f' | 'D' | 'd'
floatType
booleanLiteral
                 ::= 'true' | 'false'
characterLiteral ::= ''' (printableChar | charEscapeSeq) '''
                 ::= '"' {stringElement} '"'
stringLiteral
                      """, multiLineChars """,
                 ::= printableChar \ ('"' | '\')
stringElement
                  | UnicodeEscape
                   | charEscapeSeq
                 ::= {['"'] ['"'] char \ '"'} {'"'}
multiLineChars
processedStringLiteral
                 ::= alphaid '"' {printableChar \ ('"' | '$') | escape} '"'
                      alphaid '""" {['"'] ['"'] char \ ('"' | '$') | escape} {'"'}
                 ::= '$$'
escape
                      '$' letter { letter | digit }
                  '{' Block [';' whiteSpace stringFormat whiteSpace] '}'
                      {printableChar \ ('"', | '}', | ' ', | '\t', | '\n')}
stringFormat
                 ::=
                 ::= ''' plainid // until 2.13
symbolLiteral
                 ::=
                      '/*' "any sequence of characters; nested comments are allowed
comment
                      '//' "any sequence of characters up to end of line"
                      "new line character"
nl
                 ::=
                 ::= ';' | nl {nl}
semi
                 ::= ": at end of line that can start a template body"
colonEol
```

9.0.1 Keywords

9.0.1.1 Regular keywords

abstract	case	catch	class	def	do	else
enum	export	extends	false	final	finally	for
given	if	implicit	import	lazy	${\tt match}$	new
null	object	override	package	private	protected	return
sealed	super	then	throw	trait	true	try
type	val	var	while	with	yield	
:	=	<-	=>	<:	:>	#
@	=>>	?=>				

9.0.1.2 Soft keywords

derives end extension infix inline opaque open transparent using | * +

See the separate section on soft keywords for additional details on where a soft keyword is recognized.

9.0.2 Context-free Syntax

The context-free syntax of Scala is given by the following EBNF grammar:

9.0.2.1 Literals and Paths

```
SimpleLiteral
                  ::= ['-'] integerLiteral
                     ['-'] floatingPointLiteral
                     booleanLiteral
                      characterLiteral
                     stringLiteral
Literal
                  ::= SimpleLiteral
                    | processedStringLiteral
                      symbolLiteral
                      'null'
                  ::= id {'.' id}
QualId
                  ::= id {',' id}
ids
                  ::= id
SimpleRef
                      [id '.'] 'this'
                   [id '.'] 'super' [ClassQualifier] '.' id
                 ::= '[' id ']'
ClassQualifier
9.0.2.2 Types
                  ::= FunType
Type
                   | HkTypeParamClause '=>>' Type
                      FunParamClause '=>>' Type
                   MatchType
                   InfixType
                  ::= FunArgTypes ('=>' | '?=>') Type
FunType
                    | HKTypeParamClause '=>' Type
FunArgTypes
                  ::= InfixType
                     '(' [ FunArgType {',' FunArgType } ] ')'
                   FunParamClause
FunParamClause
                  ::= '(' TypedFunParam {',' TypedFunParam } ')'
TypedFunParam
                  ::= id ':' Type
                  ::= InfixType `match` '{' TypeCaseClauses '}'
MatchType
                  ::= RefinedType {id [nl] RefinedType}
InfixType
                  ::= AnnotType {[nl] Refinement}
RefinedType
AnnotType
                  ::= SimpleType {Annotation}
```

```
::= SimpleLiteral
SimpleType
                   | '?' TypeBounds
                   1
                     id
                   | Singleton '.' id
                   | Singleton '.' 'type'
                     ((' Types ')'
                   | Refinement
                     '$' '{' Block '}'
                   | SimpleType1 TypeArgs
                   | SimpleType1 '#' id
                 ::= SimpleRef
Singleton
                  | SimpleLiteral
                   | Singleton '.' id
            ::= Type
FunArgType
                  | '=>' Type
                 ::= ['=>'] ParamValueType
ParamType
ParamValueType
                 ::= Type ['*']
                 ::= '[' Types ']'
TypeArgs
                 ::= '{' [RefineDcl] {semi [RefineDcl]} '}'
Refinement
TypeBounds
                 ::= ['>:' Type] ['<:' Type]
TypeParamBounds ::= TypeBounds {':' Type}
                 ::= Type {',' Type}
Types
9.0.2.3 Expressions
                 ::= FunParams ('=>' | '?=>') Expr
Expr
                  | Expr1
                 ::= FunParams ('=>' | '?=>') Block
BlockResult
                   | Expr1
FunParams
                 ::= Bindings
                   id
                      ( )
                   Expr1
                 ::= ['inline'] 'if' '(' Expr ')' {nl} Expr [[semi] 'else' Expr]
                   ['inline'] 'if' Expr 'then' Expr [[semi] 'else' Expr]
                     'while' '(' Expr ')' {nl} Expr
                      'while' Expr 'do' Expr
                     'try' Expr Catches ['finally' Expr]
                     'try' Expr ['finally' Expr]
                     'throw' Expr
                     'return' [Expr]
                   | ForExpr
                   | HkTypeParamClause '=>' Expr
                   [SimpleExpr '.'] id '=' Expr
                   | SimpleExpr1 ArgumentExprs '=' Expr
                   | PostfixExpr [Ascription]
                      'inline' InfixExpr MatchClause
```

```
Ascription
                  ::= ':' InfixType
                   ':' Annotation {Annotation}
Catches
                  ::= 'catch' (Expr | ExprCaseClause)
                  ::= InfixExpr [id]
PostfixExpr
InfixExpr
                  ::= PrefixExpr
                    | InfixExpr id [nl] InfixExpr
                    | InfixExpr MatchClause
                  ::= 'match' '{' CaseClauses '}'
MatchClause
                  ::= ['-' | '+' | '~' | '!'] SimpleExpr
PrefixExpr
                  ::= SimpleRef
SimpleExpr
                      Literal
                      ٠,
                      BlockExpr
                      '$' '{' Block '}'
                      Quoted
                      quoteId
                       'new' ConstrApp {'with' ConstrApp} [[colonEol] TemplateBody
                       'new' [colonEol] TemplateBody
                       '(' ExprsInParens ')'
                       SimpleExpr '.' id
                       SimpleExpr '.' MatchClause
                       SimpleExpr TypeArgs
                       SimpleExpr ArgumentExprs
                       ''' '{' Block '}'
Quoted
                  : :=
                    ''' '[' Type ']'
ExprsInParens
                  ::=
                       ExprInParens {',' ExprInParens}
                      PostfixExpr ':' Type
ExprInParens
                  ::=
                    Expr
                  ::= '(' ['using'] ExprsInParens ')'
ParArgumentExprs
                      '(' [ExprsInParens ','] PostfixExpr ':' '_' '*' ')'
                   ArgumentExprs
                  ::= ParArgumentExprs
                    | BlockExpr
BlockExpr
                  ::= '{' (CaseClauses | Block) '}'
Block
                  ::= {BlockStat semi} [BlockResult]
BlockStat
                  ::= Import
                      {Annotation {nl}} ['implicit' | 'lazy'] Def
                      {Annotation {nl}} {LocalModifier} TmplDef
                      Extension
                      Expr1
                      EndMarker
                  ::= 'for' ('(' Enumerators ')' | '{' Enumerators '}') {nl} ['yie
ForExpr
                    | 'for' Enumerators ('do' Expr | 'yield' Expr)
                  ::= Generator {semi Enumerator | Guard}
Enumerators
                  ::= Generator
Enumerator
                    Guard
```

```
| Pattern1 '=' Expr
                  ::= ['case'] Pattern1 '<-' Expr
Generator
                  ::= 'if' PostfixExpr
Guard
CaseClauses
                 ::= CaseClause { CaseClause }
                 ::= 'case' Pattern [Guard] '=>' Block
CaseClause
                 ::= 'case' Pattern [Guard] '=>' Expr
ExprCaseClause
TypeCaseClauses
                 ::= TypeCaseClause { TypeCaseClause }
                 ::= 'case' InfixType '=>' Type [nl]
TypeCaseClause
                 ::= Pattern1 { '|' Pattern1 }
Pattern
                 ::= Pattern2 [':' RefinedType]
Pattern1
                 ::= [id '@'] InfixPattern
Pattern2
InfixPattern
                 ::= SimplePattern { id [n1] SimplePattern }
                  ::= PatVar
SimplePattern
                   | Literal
                      '(' [Patterns] ')'
                    | Quoted
                     SimplePattern1 [TypeArgs] [ArgumentPatterns]
                      'given' RefinedType
SimplePattern1
                 ::= SimpleRef
                    | SimplePattern1 '.' id
PatVar
                  ::= varid
                   Patterns
                  ::= Pattern {',' Pattern}
                 ::= '(' [Patterns] ')'
ArgumentPatterns
                      '(' [Patterns ','] Pattern2 ':' '_' '*' ')'
9.0.2.4 Type and Value Parameters
ClsTypeParamClause::= '[' ClsTypeParam {',' ClsTypeParam} ']'
                  ::= {Annotation} ['+' | '-'] id [HkTypeParamClause] TypeParamBou
ClsTypeParam
DefTypeParamClause::= '[' DefTypeParam {',' DefTypeParam} ']'
DefTypeParam
                 ::= {Annotation} id [HkTypeParamClause] TypeParamBounds
TypTypeParamClause::= '[' TypTypeParam {',' TypTypeParam} ']'
                 ::= {Annotation} id [HkTypeParamClause] TypeBounds
TypTypeParam
HkTypeParamClause ::= '[' HkTypeParam {',' HkTypeParam} ']'
                 ::= {Annotation} ['+' | '-'] (id [HkTypeParamClause] | '_') Type
HkTypeParam
ClsParamClauses
                 ::= {ClsParamClause} [[nl] '(' ['implicit'] ClsParams ')']
                 ::= [nl] '(' ClsParams ')'
ClsParamClause
                  | [nl] '(' 'using' (ClsParams | Types) ')'
                 ::= ClsParam {',' ClsParam}
ClsParams
                 ::= {Annotation} [{Modifier} ('val' | 'var') | 'inline'] Param
ClsParam
```

```
Param
                 ::= id ':' ParamType ['=' Expr]
DefParamClauses ::= {DefParamClause} [[n1] '(' ['implicit'] DefParams ')']
DefParamClause ::= [nl] '(' DefParams ')' | UsingParamClause
UsingParamClause ::= [nl] '(' 'using' (DefParams | Types) ')'
                 ::= DefParam {',' DefParam}
DefParams
                 ::= {Annotation} ['inline'] Param
DefParam
9.0.2.5 Bindings and Imports
                 ::= '(' [Binding {', 'Binding}] ')'
Bindings
                 ::= (id | ' ') [':' Type]
Binding
Modifier
                 ::= LocalModifier
                   | AccessModifier
                     'override'
                      'opaque'
                 ::= 'abstract'
LocalModifier
                   | 'final'
                      'sealed'
                      'open'
                      'implicit'
                      'lazy'
                      'inline'
                 ::= ('private' | 'protected') [AccessQualifier]
AccessModifier
                 ::= '[' id ']'
AccessQualifier
Annotation
                 ::= '@' SimpleType1 {ParArgumentExprs}
Import
                 ::= 'import' ImportExpr {',' ImportExpr}
                 ::= SimpleRef {'.' id} '.' ImportSpec
ImportExpr
                 ::= id
ImportSpec
                   ( )
                   | 'given'
                   | '{' ImportSelectors) '}'
                 ::= id ['=>' id | '=>' '] [',' ImportSelectors]
ImportSelectors
                  | WildCardSelector {',' WildCardSelector}
WildCardSelector
                 ::= 'given' [InfixType]
                      'export' ImportExpr {',' ImportExpr}
Export
                 ::=
EndMarker
                 ::= 'end' EndMarkerTag
                                          -- when followed by EOL
EndMarkerTag
                 ::= id | 'if' | 'while' | 'for' | 'match' | 'try'
```

9.0.2.6 Declarations and Definitions

'new' | 'this' | 'given' | 'extension' | 'val'

```
RefineDcl
                  ::= 'val' ValDcl
                    'def' DefDcl
                      'type' {nl} TypeDcl
Dcl
                  ::= RefineDcl
                    | 'var' VarDcl
                  ::= ids ':' Type
ValDcl
VarDcl
                       ids ':' Type
                  ::=
DefDcl
                  ::= DefSig ':' Type
                       id [DefTypeParamClause] DefParamClauses
DefSig
                  ::=
                       id [TypeParamClause] {FunParamClause} TypeBounds ['=' Type]
TypeDcl
                  ::=
                  ::= 'val' PatDef
Def
                     'var' VarDef
                      'def' DefDef
                      'type' {nl} TypeDcl
                    | TmplDef
PatDef
                       ids [':' Type] '=' Expr
                  ::=
                       Pattern2 [':' Type] '=' Expr
                  ::= PatDef
VarDef
                      ids ':' Type '=' ',
                  ::= DefSig [':' Type] '=' Expr
DefDef
                       'this' DefParamClause DefParamClauses '=' ConstrExpr
                  ::= (['case'] 'class' | 'trait') ClassDef
TmplDef
                    ['case'] 'object' ObjectDef
                      'enum' EnumDef
                    | 'given' GivenDef
ClassDef
                  ::= id ClassConstr [Template]
ClassConstr
                  ::= [ClsTypeParamClause] [ConstrMods] ClsParamClauses
ConstrMods
                  ::= {Annotation} [AccessModifier]
                  ::= id [Template]
ObjectDef
EnumDef
                  ::= id ClassConstr InheritClauses [colonEol] EnumBody
                       [GivenSig] (Type ['=' Expr] | StructuralInstance)
GivenDef
                  ::=
                  ::= [id] [DefTypeParamClause] {UsingParamClause} ':'
GivenSig
StructuralInstance ::= ConstrApp {'with' ConstrApp} 'with' TemplateBody
                  ::= 'extension' [DefTypeParamClause] '(' DefParam ')'
Extension
                       {UsingParamClause}] ExtMethods
ExtMethods
                  ::= ExtMethod | [nl] '{' ExtMethod {semi ExtMethod '}'
                  ::= {Annotation [nl]} {Modifier} 'def' DefDef
ExtMethod
Template
                  ::= InheritClauses [colonEol] [TemplateBody]
                  ::= ['extends' ConstrApps] ['derives' QualId {',' QualId}]
InheritClauses
                  ::= ConstrApp ({',' ConstrApp} | {'with' ConstrApp})
ConstrApps
                  ::= SimpleType1 {Annotation} {ParArgumentExprs}
ConstrApp
                  ::= SelfInvocation
ConstrExpr
                       '{' SelfInvocation {semi BlockStat} '}'
                    1
                       'this' ArgumentExprs {ArgumentExprs}
SelfInvocation
```

```
TemplateBody
                 ::= [nl] '{' [SelfType] TemplateStat {semi TemplateStat} '}'
TemplateStat
                      Import
                 ::=
                   Export
                     {Annotation [nl]} {Modifier} Def
                   | {Annotation [nl]} {Modifier} Dcl
                   Extension
                   | Expr1
                   | EndMarker
SelfType
                 ::= id [':' InfixType] '=>'
                   'this' ':' InfixType '=>'
                 ::= [nl] '{' [SelfType] EnumStat {semi EnumStat} '}'
EnumBody
EnumStat
                 ::= TemplateStat
                  | {Annotation [nl]} {Modifier} EnumCase
EnumCase
                 ::= 'case' (id ClassConstr ['extends' ConstrApps]] | ids)
TopStatSeq
                 ::= TopStat {semi TopStat}
TopStat
                 ::= Import
                   | Export
                   | {Annotation [nl]} {Modifier} Def
                   | Extension
                   | Packaging
                   | PackageObject
                      EndMarker
Packaging
                 ::= 'package' QualId [nl | colonEol] '{' TopStatSeq '}'
                 ::= 'package' 'object' ObjectDef
PackageObject
CompilationUnit
                 ::= {'package' QualId semi} TopStatSeq
```

10 Appendix

10.1 Context Functions - More Details

10.1.1 Syntax

Context function types associate to the right, e.g. S ?=> T ?=> U is the same as S ?=> (T ?=> U).

10.1.2 Implementation

Context function types are shorthands for class types that define apply methods with context parameters. Specifically, the N-ary function type T1, ..., TN \Rightarrow R is a shorthand for the class type ContextFunctionN[T1, ..., TN, R]. Such class types are assumed to have the following definitions, for any value of N \Rightarrow 1:

```
package scala
trait ContextFunctionN[-T1 , ... , -TN, +R]:
    def apply(using x1: T1 , ... , xN: TN): R
```

Context function types erase to normal function types, so these classes are generated on the fly for typechecking, but not realized in actual code.

Context function literals (x1: T1, ..., xn: Tn) ?=> e map context parameters xi of types Ti to the result of evaluating the expression e. The scope of each context parameter xi is e. The parameters must have pairwise distinct names.

If the expected type of the context function literal is of the form scala.ContextFunctionN[S1, ..., Sn, R], the expected type of e is R and the type Ti of any of the parameters xi can be omitted, in which case Ti = Si is assumed. If the expected type of the context function literal is some other type, all context parameter types must be explicitly given, and the expected type of e is undefined. The type of the context function literal is scala.ContextFunctionN[S1, ...,Sn, T], where T is the widened type of e. T must be equivalent to a type which does not refer to any of the context parameters xi.

The context function literal is evaluated as the instance creation expression

```
new scala.ContextFunctionN[T1, ..., Tn, T]:
    def apply(using x1: T1, ..., xn: Tn): T = e
```

A context parameter may also be a wildcard represented by an underscore _. In that case, a fresh name for the parameter is chosen arbitrarily.

Note: The closing paragraph of the Anonymous Functions section of Scala 2.13 is subsumed by context function types and should be removed.

Context function literals (x1: T1, ..., xn: Tn) ?=> e are automatically created for any expression e whose expected type is scala.ContextFunctionN[T1, ..., Tn, R], unless e is itself a context function literal. This is analogous to the automatic insertion of scala.FunctionO around expressions in by-name argument position.

Context function types generalize to $\mathbb{N} > 22$ in the same way that function types do, see the corresponding documentation.

10.1.3 Examples

See the section on Expressiveness from Simplicitly: foundations and applications of implicit function types.

10.1.3.1 Type Checking After desugaring no additional typing rules are required for context function types.

10.2 Opaque Type Aliases: More Details

10.2.0.1 Syntax

```
Modifier ::= ... | 'opaque'
```

opaque is a soft modifier. It can still be used as a normal identifier when it is not in front of a definition keyword.

Opaque type aliases must be members of classes, traits, or objects, or they are defined at the top-level. They cannot be defined in local blocks.

10.2.0.2 Type Checking The general form of a (monomorphic) opaque type alias is

```
opaque type T >: L <: U = R
```

where the lower bound L and the upper bound U may be missing, in which case they are assumed to be ${\tt scala.Nothing}$ and ${\tt scala.Any}$, respectively. If bounds are given, it is checked that the right hand side R conforms to them, i.e. L <: R and R <: U. F-bounds are not supported for opaque type aliases: T is not allowed to appear in L or U.

Inside the scope of the alias definition, the alias is transparent: T is treated as a normal alias of R. Outside its scope, the alias is treated as the abstract type

```
type T >: L <: U
```

A special case arises if the opaque type alias is defined in an object. Example:

```
object o:
   opaque type T = R
```

In this case we have inside the object (also for non-opaque types) that o.T is equal to T or its expanded form o.this.T. Equality is understood here as mutual subtyping, i.e. o.T <: o.this.T and o.this.T <: T. Furthermore, we have by the rules of opaque type aliases that o.this.T equals R. The two equalities compose. That is, inside o, it is also known that o.T is equal to R. This means the following code type-checks:

```
object o:
   opaque type T = Int
   val x: Int = id(2)
def id(x: o.T): o.T = x
```

10.2.0.3 Type Parameters of Opaque Types Opaque type aliases can have a single type parameter list. The following aliases are well-formed

```
opaque type F[T] = (T, T)
opaque type G = [T] =>> List[T]
but the following are not:
opaque type BadF[T] = [U] =>> (T, U)
opaque type BadG = [T] =>> [U] => (T, U)
```

10.2.0.4 Translation of Equality Comparing two values of opaque type with == or != normally uses universal equality, unless another overloaded == or != operator is defined for the type. To avoid boxing, the operation is mapped after type checking to the (in-)equality operator defined on the underlying type. For instance,

```
opaque type T = Int
...
val x: T
val y: T
x == y  // uses Int equality for the comparison.
```

10.2.0.5 Top-level Opaque Types An opaque type alias on the top-level is transparent in all other top-level definitions in the sourcefile where it appears, but is opaque in nested objects and classes and in all other source files. Example:

```
// in test1.scala
opaque type A = String
val x: A = "abc"

object obj:
   val y: A = "abc" // error: found: "abc", required: A

// in test2.scala
def z: String = x // error: found: A, required: String
```

This behavior becomes clear if one recalls that top-level definitions are placed in their own synthetic object. For instance, the code in test1.scala would expand to

```
object test1$package:
```

```
opaque type A = String
  val x: A = "abc"

object obj:
  val y: A = "abc" // error: cannot assign "abc" to opaque type alias A
```

The opaque type alias A is transparent in its scope, which includes the definition of x, but not the definitions of obj and y.

10.2.0.6 Relationship to SIP 35 Opaque types in Scala 3 are an evolution from what is described in Scala SIP 35.

The differences compared to the state described in this SIP are:

- 1. Opaque type aliases cannot be defined anymore in local statement sequences.
- 2. The scope where an opaque type alias is visible is now the whole scope where it is defined, instead of just a companion object.
- 3. The notion of a companion object for opaque type aliases has been dropped.
- 4. Opaque type aliases can have bounds.
- 5. The notion of type equality involving opaque type aliases has been clarified. It was strengthened with respect to the previous implementation of SIP 35.

10.3 Named Type Arguments - More Details 🗹

10.3.1 Syntax

The addition to the grammar is:

```
SimpleExpr1 ::= ...

| SimpleExpr (TypeArgs | NamedTypeArgs)
NamedTypeArgs ::= '[' NamedTypeArg {',' NamedTypeArg} ']'
NamedTypeArg ::= id '=' Type
```

Note in particular that named arguments cannot be passed to type constructors:

```
class C[T]
```

```
val x: C[T = Int] = // error
  new C[T = Int] // error

class E extends C[T = Int] // error
```

10.3.2 Compatibility considerations

Named type arguments do not have an impact on binary compatibility, but they have an impact on source compatibility: if the name of a method type parameter

is changed, any existing named reference to this parameter will break. This means that the names of method type parameters are now part of the public API of a library.

(Unimplemented proposal: to mitigate this, scala.deprecatedName could be extended to also be applicable on method type parameters.)

10.4 Parameter Untupling - More Details

10.4.0.1 Motivation Say you have a list of pairs

```
val xs: List[(Int, Int)]
```

and you want to map **xs** to a list of **Ints** so that each pair of numbers is mapped to their sum. Previously, the best way to do this was with a pattern-matching decomposition:

```
xs.map {
   case (x, y) => x + y
}
```

While correct, this is inconvenient. Instead, we propose to write it the following way:

```
xs.map {
    (x, y) => x + y
}
```

or, equivalently:

```
xs.map(_ + _)
```

Generally, a function value with n>1 parameters can be converted to a function with tupled arguments if the expected type is a unary function type of the form ((T 1, ..., T n)) \Rightarrow U.

10.4.0.2 Type Checking Let a function f of the form (p1, ..., pn) => e for n != 1, parameters p1, ..., pn, and an expression e.

If the expected type of f is a fully defined function type or SAM-type that has a single parameter of a subtype of ProductN[T1, ..., Tn], where each type Ti fits the corresponding parameter pi. Then f will conform to the function type ProductN[T1, ..., Tn] => R.

A type Ti fits a parameter pi if one of the following two cases is true:

- pi comes without a type, i.e. it is a simple identifier or _.
- pi is of the form x: Ui or _: Ui and Ti <: Ui.

Auto-tupling composes with eta-expansion. That is an n-ary function generated by eta-expansion can in turn be adapted to the expected type with auto-tupling.

10.4.0.2.1 Term adaptation If the function

```
(p1: T1, ..., pn: Tn) => e
is typed as ProductN[T1, ..., Tn] => Te, then it will be transformed to
(x: TupleN[T1, ..., Tn]) =>
  def p1: T1 = x._1
    ...
  def pn: Tn = x._n
  e
```

Generic tuples

If we come to support generic tuples, which provide the possibility of having tuples/functions of arities larger than 22 we would need to additionally support generic tuples of the form T1 *: T2 *: Translation of such a tuples would use the apply method on the tuple to access the elements instead of the _N methods of Product.

10.4.0.3 Migration Code like this could not be written before, hence the new notation would not be ambiguous after adoption.

Though it is possible that someone has written an implicit conversion form (T1, ..., Tn) => R to TupleN[T1, ..., Tn] => R for some n. This change could be detected and fixed by Scalafix. Furthermore, such conversion would probably be doing the same translation (semantically) but in a less efficient way.

10.4.0.4 Reference For more information, see Issue #897.

10.5 Dropped: Class Shadowing - More Details

Spec diff: in section 5.1.4 Overriding, add M' must not be a class.

Why do we want to make this change to the language?

Class shadowing is irregular compared to other types of overrides. Indeed, inner classes are not actually overriden but simply shadowed.

How much existing code is going to be affected?

From all the code compiled so far with Scala 3 the only instance of this I could find is in the stdlib. Looking at this commit it seems like the usage of class shadowing was accidental.

How exactly is existing code going to be affected?

Code that relies on overridden inner classes will stop compiling.

Is this change going to be migratable automatically?

No.

10.6 Dropped: Weak Conformance - More Details

To simplify the underlying type theory, Scala 3 drops the notion of weak conformance altogether. Instead, it provides more flexibility when assigning a type to a constant expression. The new rule is:

- If a list of expressions Es appears as one of
 - the elements of a vararg parameter, or
 - the alternatives of an if-then-else or match expression, or
 - the body and catch results of a try expression,

and all expressions have primitive numeric types, but they do not all have the same type, then the following is attempted:

- the expressions Es are partitioned into Int constants on the one hand,
 and all other expressions on the other hand,
- if all the other expressions have the same numeric type T (which can be one of Byte, Short, Char, Int, Long, Float, Double), possibly after widening, and if none of the Int literals would incur a loss of precision when converted to T, then they are thus converted (the other expressions are left unchanged regardless),
- otherwise, the expressions Es are used unchanged.

A loss of precision occurs for an Int -> Float conversion of a constant c if c.toFloat.toInt != c. For an Int -> Byte conversion it occurs if c.toByte.toInt != c. For an Int -> Short conversion, it occurs if c.toShort.toInt != c.

10.6.1 Examples

```
inline val b = 33
def f(): Int = b + 1
Array(b, 33, 5.5)
                       : Array[Double] // b is an inline val
Array(f(), 33, 5.5)
                       : Array[AnyVal] // f() is not a constant
Array(5, 11L)
                       : Array[Long]
Array(5, 11L, 5.5)
                       : Array[AnyVal] // Long and Double found
Array(1.0f, 2)
                       : Array[Float]
Array(1.0f, 1234567890): Array[AnyVal] // loss of precision
Array(b, 33, 'a')
                       : Array[Char]
Array(5.toByte, 11)
                       : Array[Byte]
```

10.7 Automatic Eta Expansion - More Details

10.7.1 Motivation

Scala maintains a convenient distinction between *methods* and *functions*. Methods are part of the definition of a class that can be invoked in objects while functions are complete objects themselves, making them first-class entities. For example, they can be assigned to variables. These two mechanisms are bridged in Scala by a mechanism

called *eta-expansion* (also called eta-abstraction), which converts a reference to a method into a function. Intuitively, a method m can be passed around by turning it into an object: the function $x \Rightarrow m(x)$.

In this snippet which assigns a method to a val, the compiler will perform *automatic* eta-expansion, as shown in the comment:

```
def m(x: Int, y: String) = ???
val f = m // becomes: val f = (x: Int, y: String) => m(x, y)
```

In Scala 2, a method reference m was converted to a function value only if the expected type was a function type, which means the conversion in the example above would not have been triggered, because val f does not have a type ascription. To still get eta-expansion, a shortcut m _ would force the conversion.

For methods with one or more parameters like in the example above, this restriction has now been dropped. The syntax ${\tt m}$ is no longer needed and will be deprecated in the future.

10.7.2 Automatic eta-expansion and partial application

In the following example m can be partially applied to the first two parameters. Assignining m to f1 will automatically eta-expand.

```
def m(x: Boolean, y: String)(z: Int): List[Int]
val f1 = m
val f2 = m(true, "abc")
This creates two function values:
f1: (Boolean, String) => Int => List[Int]
f2: Int => List[Int]
```

10.7.3 Automatic eta-expansion and implicit parameter lists

Methods with implicit parameter lists will always get applied to implicit arguments.

```
def foo(x: Int)(implicit p: Double): Float = ???
implicit val bla: Double = 1.0
val bar = foo // val bar: Int => Float = ...
```

10.7.4 Automatic Eta-Expansion and query types

A method with context parameters can be expanded to a value of a context type by writing the expected context type explicitly.

```
def foo(x: Int)(using p: Double): Float = ???
val bar: Double ?=> Float = foo(3)
```

10.7.5 Rules

- If m has an argument list with one or more parameters, we always eta-expand
- If m is has an empty argument list (i.e. has type ()R):
 - 1. If the expected type is of the form () => T, we eta expand.
 - 2. If m is defined by Java, or overrides a Java defined method, we insert ().
 - 3. Otherwise we issue an error of the form:

Thus, an unapplied method with an empty argument list is only converted to a function when a function type is expected. It is considered best practice to either explicitly apply the method to (), or convert it to a function with () \Rightarrow m().

The method value syntax m _ is deprecated.

10.7.6 Reference

For more information, see PR #2701.

10.8 Implicit Conversions - More Details

10.8.1 Implementation

An implicit conversion, or view, from type S to type T is defined by either:

- An implicit def which has type S => T or (=> S) => T
- An implicit value which has type Conversion[S, T]

The standard library defines an abstract class Conversion:

```
package scala
@java.lang.FunctionalInterface
abstract class Conversion[-T, +U] extends Function1[T, U]:
    def apply(x: T): U
```

Function literals are automatically converted to Conversion values.

Views are applied in three situations:

- 1. If an expression e is of type T, and T does not conform to the expression's expected type pt. In this case, an implicit v which is applicable to e and whose result type conforms to pt is searched. The search proceeds as in the case of implicit parameters, where the implicit scope is the one of $T \Rightarrow pt$. If such a view is found, the expression e is converted to v(e).
- 2. In a selection e.m with e of type T, if the selector m does not denote an accessible member of T. In this case, a view v which is applicable to e and whose result contains an accessible member named m is searched. The search proceeds as in the case of implicit parameters, where the implicit scope is the one of T. If such a view is found, the selection e.m is converted to v(e).m.
- 3. In an application e.m(args) with e of type T, if the selector m denotes some accessible member(s) of T, but none of these members is applicable to the arguments args. In this case, a view v which is applicable to e and whose result contains a method m which is applicable to args is searched. The search

proceeds as in the case of implicit parameters, where the implicit scope is the one of T. If such a view is found, the application e.m(args) is converted to v(e).m(args).

10.9 Differences with Scala 2 implicit conversions

In Scala 2, views whose parameters are passed by-value take precedence over views whose parameters are passed by-name. This is no longer the case in Scala 3. A type error reporting the ambiguous conversions will be emitted in cases where this rule would be applied in Scala 2:

In Scala 2, implicit values of a function type would be considered as potential views. In Scala 3, these implicit value need to have type Conversion:

```
// Scala 2:
def foo(x: Int)(implicit conv: Int => String): String = x

// Becomes with Scala 3:
def foo(x: Int)(implicit conv: Conversion[Int, String]): String = x

// Call site is unchanged:
foo(4)(_.toString)

// Scala 2:
implicit val myConverter: Int => String = _.toString

// Becomes with Scala 3:
implicit val myConverter: Conversion[Int, String] = _.toString
```

Note that implicit conversions are also affected by the changes to implicit resolution between Scala 2 and Scala 3.

10.9.1 Motivation for the changes

The introduction of Conversion in Scala 3 and the decision to restrict implicit values of this type to be considered as potential views comes from the desire to remove surprising behavior from the language:

This snippet contains a type error. The right hand side of val x does not conform to type String. In Scala 2, the compiler will use m as an implicit conversion from Int to String, whereas Scala 3 will report a type error, because Map isn't an instance of Conversion.

10.9.2 Migration path

Implicit values that are used as views should see their type changed to Conversion.

For the migration of implicit conversions that are affected by the changes to implicit resolution, refer to the Changes in Implicit Resolution for more information.

10.9.3 Reference

For more information about implicit resolution, see Changes in Implicit Resolution. Other details are available in PR #2065.

10.10 Programmatic Structural Types - More Details 🗹

10.10.1 Syntax

```
SimpleType ::= ... | Refinement
Refinement ::= '{' RefineStatSeq '}'
RefineStatSeq ::= RefineStat {semi RefineStat}
RefineStat ::= 'val' VarDcl | 'def' DefDcl | 'type' {nl} TypeDcl
```

10.10.2 Implementation of structural types

The standard library defines a universal marker trait Selectable in the package scala:

trait Selectable extends Any

An implementation of Selectable that relies on Java reflection is available in the standard library: scala.reflect.Selectable. Other implementations can be envisioned for platforms where Java reflection is not available.

Implementations of Selectable have to make available one or both of the methods selectDynamic and applyDynamic. The methods could be members of the Selectable implementation or they could be extension methods.

The selectDynamic method takes a field name and returns the value associated with that name in the Selectable. It should have a signature of the form:

```
def selectDynamic(name: String): T
```

Often, the return type T is Any.

Unlike scala.Dynamic, there is no special meaning for an updateDynamic method. However, we reserve the right to give it meaning in the future. Consequently, it is recommended not to define any member called updateDynamic in Selectables.

The applyDynamic method is used for selections that are applied to arguments. It takes a method name and possibly Classes representing its parameters types as well as the arguments to pass to the function. Its signature should be of one of the two following forms:

```
def applyDynamic(name: String)(args: Any*): T
def applyDynamic(name: String, ctags: Class[?]*)(args: Any*): T
```

Both versions are passed the actual arguments in the args parameter. The second version takes in addition a vararg argument of <code>java.lang.Classes</code> that identify the method's parameter classes. Such an argument is needed if applyDynamic is implemented using Java reflection, but it could be useful in other cases as well. <code>selectDynamic</code> and applyDynamic can also take additional context parameters in using clauses. These are resolved in the normal way at the callsite.

Given a value v of type C { Rs }, where C is a class reference and Rs are structural refinement declarations, and given v.a of type U, we consider three distinct cases:

• If U is a value type, we map v.a to:

```
v.selectDynamic("a").asInstanceOf[U]
```

• If U is a method type (T11, ..., T1n)...(TN1, ..., TNn): R and it is not a dependent method type, we map v.a(a11, ..., a1n)...(aN1, ..., aNn) to:

```
v.applyDynamic("a")(a11, ..., a1n, ..., aN1, ..., aNn)
.asInstanceOf[R]
```

If this call resolves to an applyDynamic method of the second form that takes a Class[?] * argument, we further rewrite this call to

```
v.applyDynamic("a", c11, ..., c1n, ..., cN1, ... cNn)(
  a11, ..., a1n, ..., aN1, ..., aNn)
  .asInstanceOf[R]
```

where each c_ij is the literal java.lang.Class[?] of the type of the formal parameter Tij, i.e., classOf[Tij].

• If U is neither a value nor a method type, or a dependent method type, an error is emitted.

Note that v's static type does not necessarily have to conform to Selectable, nor does it need to have selectDynamic and applyDynamic as members. It suffices that there is an implicit conversion that can turn v into a Selectable, and the selection methods could also be available as extension methods.

10.10.3 Limitations of structural types

- Dependent methods cannot be called via structural call.
- Overloaded methods cannot be called via structural call.
- Refinements do not handle polymorphic methods.

10.10.4 Differences with Scala 2 structural types

- Scala 2 supports structural types by means of Java reflection. Unlike Scala 3, structural calls do not rely on a mechanism such as Selectable, and reflection cannot be avoided.
- In Scala 2, structural calls to overloaded methods are possible.
- In Scala 2, mutable vars are allowed in refinements. In Scala 3, they are no longer allowed.

10.10.5 Context

For more information, see Rethink Structural Types.

10.11 Dependent Function Types - More Details

Initial implementation in PR #3464.

10.11.1 Syntax

```
FunArgTypes ::= InfixType
| '(' [ FunArgType {',' FunArgType } ] ')'
| '(' TypedFunParam {',' TypedFunParam } ')'

TypedFunParam ::= id ':' Type

Dependent function types associate to the right, e.g. (s: S) (t: T) U is the
```

Dependent function types associate to the right, e.g. (s: S) (t: T) U is the same as (s: S) ((t: T) U).

10.11.2 Implementation

Dependent function types are shorthands for class types that define apply methods with a dependent result type. Dependent function types desugar to refinement types of scala.FunctionN. A dependent function type $(x1: K1, ..., xN: KN) \Rightarrow R$ of arity N translates to:

```
FunctionN[K1, ..., Kn, R'] with
  def apply(x1: K1, ..., xN: KN): R
```

where the result type parameter R' is the least upper approximation of the precise result type R without any reference to value parameters x1, ..., xN.

The syntax and semantics of anonymous dependent functions is identical to the one of regular functions. Eta expansion is naturally generalized to produce dependent function types for methods with dependent result types.

Dependent functions can be implicit, and generalize to arity $\mathbb{N} > 22$ in the same way that other functions do, see the corresponding documentation.

10.11.3 Examples

• depfuntype.scala

• eff-dependent.scala

10.11.3.1 Type Checking After desugaring no additional typing rules are required for dependent function types.

10.12 Intersection Types - More Details

10.12.1 Syntax

Syntactically, the type S & T is an infix type, where the infix operator is &. The operator & is a normal identifier with the usual precedence and subject to usual resolving rules. Unless shadowed by another definition, it resolves to the type scala.&, which acts as a type alias to an internal representation of intersection types.

Type ::= ...| InfixType

InfixType ::= RefinedType {id [nl] RefinedType}

10.12.2 Subtyping Rules

From the rules above, we can show that & is commutative: A & B <: B & A for any type A and B.

In another word, A & B is the same type as B & A, in the sense that the two types have the same values and are subtypes of each other.

If C is a type constructor, then C[A] & C[B] can be simplified using the following three rules:

- If C is covariant, C[A] & C[B] ~> C[A & B]
- If C is contravariant, C[A] & C[B] ~> C[A | B]
- If C is non-variant, emit a compile error

When C is covariant, C[A & B] <: C[A] & C[B] can be derived:

C[A & B] <: C[A] & C[B]

When C is contravariant, C[A | B] <: C[A] & C[B] can be derived:

A <: A	B <: B
A <: A B	B <: A B
C[A B] <: C[A]	C[A B] <: C[B]
C[A B] <:	 C[A] & C[B]

10.12.3 Erasure

The erased type for S & T is the erased glb (greatest lower bound) of the erased type of S and T. The rules for erasure of intersection types are given below in pseudocode:

```
|S \& T| = glb(|S|, |T|)
```

```
glb(JArray(A), JArray(B)) = JArray(glb(A, B))
                           = JArray(T)
glb(JArray(T), )
glb(_, JArray(T))
                           = JArray(T)
glb(A, B)
                           = A
                                                    if A extends B
                           = B
                                                    if B extends A
glb(A, B)
glb(A, _)
                           = A
                                                    if A is not a trait
glb(_, B)
                           = B
                                                    if B is not a trait
glb(A, )
                           = A
                                                    // use first
```

In the above, |T| means the erased type of T, JArray refers to the type of Java Array.

See also: TypeErasure#erasedGlb.

10.12.4 Relationship with Compound Type (with)

Intersection types A & B replace compound types A with B in Scala 2. For the moment, the syntax A with B is still allowed and interpreted as A & B, but its usage as a type (as opposed to in a new or extends clause) will be deprecated and removed in the future.

10.13 Type Lambdas - More Details 💆

10.13.1 Syntax

```
Type ::= ... | TypeParamClause '=>>' Type
TypeParamClause ::= '[' TypeParam {',' TypeParam} ']'
TypeParam ::= {Annotation} (id [HkTypeParamClause] | '_') TypeBounds
TypeBounds ::= ['>:' Type] ['<:' Type]</pre>
```

10.13.1.1 Type Checking A type lambda such as $[X] \implies F[X]$ defines a function from types to types. The parameter(s) may carry bounds. If a parameter is bounded, as in $[X >: L <: U] \implies F[X]$ it is checked that arguments to the parameters conform to the bounds L and U. Only the upper bound U can be F-bounded, i.e. X can appear in it.

10.13.2 Subtyping Rules

Assume two type lambdas

```
type TL1 = [X >: L1 <: U1] =>> R1
type TL2 = [X >: L2 <: U2] =>> R2
```

Then TL1 <: TL2, if

- the type interval L2..U2 is contained in the type interval L1..U1 (i.e. L1 <: L2 and U2 <: U1),
- R1 <: R2

Here we have relied on alpha renaming to match the two bound types X.

A partially applied type constructor such as List is assumed to be equivalent to its eta expansion. I.e, List = [X] =>> List[X]. This allows type constructors to be compared with type lambdas.

10.13.3 Relationship with Parameterized Type Definitions

A parameterized type definition

type
$$T[X] = R$$

is regarded as a shorthand for an unparameterized definition with a type lambda as right-hand side:

type
$$T = [X] \implies R$$

If the type definition carries + or - variance annotations, it is checked that the variance annotations are satisfied by the type lambda. For instance,

type
$$F2[A, +B] = A \Rightarrow B$$

expands to

type
$$F2 = [A, B] \Rightarrow A \Rightarrow B$$

and at the same time it is checked that the parameter B appears covariantly in $A \Rightarrow B$.

A parameterized abstract type

is regarded as shorthand for an unparameterized abstract type with type lambdas as bounds.

However, if L is Nothing it is not parameterized, since Nothing is treated as a bottom type for all kinds. For instance,

type
$$T[X] <: X \Rightarrow X$$

is expanded to

type T
$$>$$
: Nothing $<$: ([X] \Rightarrow > X \Rightarrow X)

instead of

type
$$T >: ([X] \Rightarrow> Nothing) <: ([X] \Rightarrow> X \Rightarrow X)$$

The same expansions apply to type parameters. For instance,

is treated as a shorthand for

Abstract types and opaque type aliases remember the variances they were created with. So the type

is known to be contravariant in A and covariant in B and can be instantiated only with types that satisfy these constraints. Likewise

```
opaque type O[X] = List[X]
```

O is known to be invariant (and not covariant, as its right hand side would suggest). On the other hand, a transparent alias

would be treated as covariant, X is used covariantly on its right-hand side.

Note: The decision to treat **Nothing** as universal bottom type is provisional, and might be changed after further discussion.

Note: Scala 2 and 3 differ in that Scala 2 also treats **Any** as universal top-type. This is not done in Scala 3. See also the discussion on kind polymorphism

10.13.4 Curried Type Parameters

The body of a type lambda can again be a type lambda. Example:

type
$$TL = [X] \implies [Y] \implies (X, Y)$$

Currently, no special provision is made to infer type arguments to such curried type lambdas. This is left for future work.

10.14 Union Types - More Details 🗹

10.14.1 Syntax

Syntactically, unions follow the same rules as intersections, but have a lower precedence, see Intersection Types - More Details.

10.14.1.1 Interaction with pattern matching syntax | is also used in pattern matching to separate pattern alternatives and has lower precedence than: as used in typed patterns, this means that:

is still equivalent to:

and not to:

10.14.2 Subtyping Rules

- A is always a subtype of A | B for all A, B.
- If A <: C and B <: C then A | B <: C
- Like &, | is commutative and associative:

$$A \mid B = := B \mid A$$

 $A \mid (B \mid C) = := (A \mid B) \mid C$

• & is distributive over |:

From these rules it follows that the *least upper bound* (LUB) of a set of types is the union of these types. This replaces the definition of least upper bound in the Scala 2 specification.

10.14.3 Motivation

The primary reason for introducing union types in Scala is that they allow us to guarantee that for every set of types, we can always form a finite LUB. This is both useful in practice (infinite lubs in Scala 2 were approximated in an ad-hoc way,

resulting in imprecise and sometimes incredibly long types) and in theory (the type system of Scala 3 is based on the DOT calculus, which has union types).

Additionally, union types are a useful construct when trying to give types to existing dynamically typed APIs, this is why they're an integral part of TypeScript and have even been partially implemented in Scala.js.

10.14.4 Join of a union type

In some situation described below, a union type might need to be widened to a non-union type, for this purpose we define the *join* of a union type T1 | ... | Tn as the smallest intersection type of base class instances of T1,...,Tn. Note that union types might still appear as type arguments in the resulting type, this guarantees that the join is always finite.

10.14.4.1 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 | B is C[A | B] & D
```

10.14.5 Type inference

When inferring the result type of a definition (val, var, or def) and the type we are about to infer is a union type, then we replace it by its join. Similarly, when instantiating a type argument, if the corresponding type parameter is not upper-bounded by a union type and the type we are about to instantiate is a union type, we replace it by its join. This mirrors the treatment of singleton types which are also widened to their underlying type unless explicitly specified. The motivation is the same: inferring types which are "too precise" can lead to unintuitive typechecking issues later on.

Note: Since this behavior limits the usability of union types, it might be changed in the future. For example by not widening unions that have been explicitly written down by the user and not inferred, or by not widening a type argument when the corresponding type parameter is covariant.

See PR #2330 and Issue #4867 for further discussions.

10.14.5.1 Example

```
import scala.collection.mutable.ListBuffer
val x = ListBuffer(Right("foo"), Left(0))
val y: ListBuffer[Either[Int, String]] = x
```

This code typechecks because the inferred type argument to ListBuffer in the right-hand side of x was Left[Int, Nothing] | Right[Nothing, String] which was widened to Either[Int, String]. If the compiler hadn't done this widening, the last line wouldn't typecheck because ListBuffer is invariant in its argument.

10.14.6 Members

The members of a union type are the members of its join.

10.14.6.1 Example The following code does not typecheck, because method hello is not a member of AnyRef which is the join of A | B.

```
trait A { def hello: String }
trait B { def hello: String }

def test(x: A | B) = x.hello // error: value `hello` is not a member of A | B

On the otherhand, the following would be allowed

trait C { def hello: String }
trait A extends C with D

trait B extends C with E

def test(x: A | B) = x.hello // ok as `hello` is a member of the join of A | B what
```

10.14.7 Exhaustivity checking

If the selector of a pattern match is a union type, the match is considered exhaustive if all parts of the union are covered.

10.14.8 Erasure

The erased type for A | B is the *erased least upper bound* of the erased types of A and B. Quoting from the documentation of TypeErasure#erasedLub, 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
 - 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.

10.15 Erased Terms Spec

10.15.1 Rules

- 1. The **erased** modifier can appear:
 - At the start of a parameter block of a method, function or class
 - In a method definition
 - In a val definition (but not lazy val or var)

```
erased val x = ...
erased def f = ...

def g(erased x: Int) = ...

(erased x: Int) => ...
  def h(x: (erased Int) => Int) = ...

class K(erased x: Int) { ... }
```

- 2. A reference to an erased definition can only be used
 - Inside the expression of argument to an erased parameter
 - Inside the body of an erased val or def
- 3. Functions
 - (erased x1: T1, x2: T2, ..., xN: TN) => y : (erased T1, T2, ..., TN) => R (given erased x1: T1, x2: T2, ..., xN: TN) => y: (given erased T1, T2, ... (given erased T1) => R <:< erased T1 => R
 - (given erased T1, T2) \Rightarrow R <:< (erased T1, T2) \Rightarrow R
 - ...

Note that there is no subtype relation between (erased T) \Rightarrow R and T \Rightarrow R (or (given erased T) \Rightarrow R and (given T) \Rightarrow R)

4. Eta expansion

```
if def f(erased x: T): U then f: (erased T) \Rightarrow U.
```

- 5. Erasure Semantics
 - All erased parameters are removed from the function
 - All argument to **erased** parameters are not passed to the function
 - All erased definitions are removed
 - All (erased T1, T2, ..., TN) => R and (given erased T1, T2, ..., TN) => R become () => R
- 6. Overloading

Method with **erased** parameters will follow the normal overloading constraints after erasure.

7. Overriding

- Member definitions overriding each other must both be erased or not be erased
- def foo(x: T): U cannot be overridden by def foo(erased x: T): U
 and vice-versa

10.16 Macros Spec ☑

10.16.1 Implementation

10.16.1.1 Syntax Compared to the Scala 3 reference grammar there are the following syntax changes:

In addition, an identifier x starting with a t that appears inside a quoted expression or type is treated as a splice x and a quoted identifier x that appears inside a splice is treated as a quote x

10.16.1.2 Implementation in scalac Quotes and splices are primitive forms in the generated abstract syntax trees. Top-level splices are eliminated during macro expansion while typing. On the other hand, top-level quotes are eliminated in an expansion phase PickleQuotes phase (after typing and pickling). PCP checking occurs while preparing the RHS of an inline method for top-level splices and in the Staging phase (after typing and before pickling).

Macro-expansion works outside-in. If the outermost scope is a splice, the spliced AST will be evaluated in an interpreter. A call to a previously compiled method can be implemented as a reflective call to that method. With the restrictions on splices that are currently in place that's all that's needed. We might allow more interpretation in splices in the future, which would allow us to loosen the restriction. Quotes in spliced, interpreted code are kept as they are, after splices nested in the quotes are expanded.

If the outermost scope is a quote, we need to generate code that constructs the quoted tree at run-time. We implement this by serializing the tree as a TASTy structure, which is stored in a string literal. At runtime, an unpickler method is called to describing the string into a tree.

Splices inside quoted code insert the spliced tree as is, after expanding any quotes in the spliced code recursively.

10.16.2 Formalization

The phase consistency principle can be formalized in a calculus that extends simply-typed lambda calculus with quotes and splices.

10.16.2.1 Syntax The syntax of terms, values, and types is given as follows:

Typing rules are formulated using a stack of environments Es. Individual environments E consist as usual of variable bindings x: T. Environments can be combined using the two combinators ' and \$.

The two environment combinators are both associative with left and right identity ().

10.16.2.2 Operational semantics: We define a small step reduction relation --> with the following rules:

The first rule is standard call-by-value beta-reduction. The second rule says that splice and quotes cancel each other out. The third rule is a context rule; it says that reduction is allowed in the hole [] position of an evaluation context. Evaluation contexts e and splice evaluation context e_s are defined syntactically as follows:

Eval context
$$e ::= [] | e t | v e | 'e_s[${e}]$$

Splice context $e_s ::= [] | (x: T) => e_s | e_s t | u e_s$

10.16.2.3 Typing rules Typing judgments are of the form Es |- t: T. There are two substructural rules which express the fact that quotes and splices cancel each other out:

The lambda calculus fragment of the rules is standard, except that we use a stack of environments. The rules only interact with the topmost environment of the stack.

The rules for quotes and splices map between expr T and T by trading ' and \$ between environments and terms.

The meta theory of a slightly simplified 2-stage variant of this calculus is studied separately.

10.16.3 Going Further

The metaprogramming framework as presented and currently implemented is quite restrictive in that it does not allow for the inspection of quoted expressions and types. It's possible to work around this by providing all necessary information as normal, unquoted inline parameters. But we would gain more flexibility by allowing for the inspection of quoted code with pattern matching. This opens new possibilities.

For instance, here is a version of power that generates the multiplications directly if the exponent is statically known and falls back to the dynamic implementation of power otherwise.

```
import scala.quoted._
inline def power(x: Double, n: Int): Double =
   ${ powerExpr('x, 'n) }
private def powerExpr(x: Expr[Double], n: Expr[Int])
                     (using Quotes): Expr[Double] =
   n.value match
      case Some(m) => powerExpr(x, m)
      case => '{ dynamicPower($x, $n) }
private def powerExpr(x: Expr[Double], n: Int)
                     (using Quotes): Expr[Double] =
   if n == 0 then '{ 1.0 }
   else if n == 1 then x
   else if n \% 2 == 0 then '{ val y = x * x; $\{ powerExpr('y, n / 2) \} }
   else '{ $x * ${ powerExpr(x, n - 1) } }
private def dynamicPower(x: Double, n: Int): Double =
   if n == 0 then 1.0
   else if n \% 2 == 0 then dynamicPower(x * x, n / 2)
   else x * dynamicPower(x, n - 1)
```

In the above, the method .value maps a constant expression of the type Expr[T] to its value of the type T.

With the right extractors, the "AsFunction" conversion that maps expressions over functions to functions over expressions can be implemented in user code:

This assumes an extractor

object Lambda:

```
def unapply[T, U](x: Expr[T => U]): Option[Expr[T] => Expr[U]]
```

Once we allow inspection of code via extractors, it's tempting to also add constructors that create typed trees directly without going through quotes. Most likely, those constructors would work over Expr types which lack a known type argument. For instance, an Apply constructor could be typed as follows:

```
def Apply(fn: Expr[Any], args: List[Expr[Any]]): Expr[Any]
```

This would allow constructing applications from lists of arguments without having to match the arguments one-by-one with the corresponding formal parameter types of the function. We then need "at the end" a method to convert an Expr[Any] to an Expr[T] where T is given from the outside. For instance, if code yields a Expr[Any], then code.atType[T] yields an Expr[T]. The atType method has to be implemented as a primitive; it would check that the computed type structure of Expr is a subtype of the type structure representing T.

Before going down that route, we should evaluate in detail the tradeoffs it presents. Constructing trees that are only verified a posteriori to be type correct loses a lot of guidance for constructing the right trees. So we should wait with this addition until we have more use-cases that help us decide whether the loss in type-safety is worth the gain in flexibility. In this context, it seems that deconstructing types is less error-prone than deconstructing terms, so one might also envisage a solution that allows the former but not the latter.

10.16.4 Conclusion

Metaprogramming has a reputation of being difficult and confusing. But with explicit Expr/Type types and quotes and splices it can become downright pleasant. A simple strategy first defines the underlying quoted or unquoted values using Expr and Type and then inserts quotes and splices to make the types line up. Phase consistency is at the same time a great guideline where to insert a splice or a quote and a vital sanity check that the result makes sense.

10.17 A Classification of Proposed Language Features

This document provides an overview of the constructs proposed for Scala 3 with the aim to facilitate the discussion what to include and when to include it. It classifies features into eight groups: (1) essential foundations, (2) simplifications, (3) restrictions, (4) dropped features, (5) changed features, (6) new features, (7) features oriented towards metaprogramming with the aim to replace existing macros, and (8) changes to type checking and inference.

Each group contains sections classifying the status (i.e. relative importance to be a part of Scala 3, and relative urgency when to decide this) and the migration cost of the constructs in it.

The current document reflects the state of things as of April, 2019. It will be updated to reflect any future changes in that status.

10.17.1 Essential Foundations

These new constructs directly model core features of DOT, higher-kinded types, and the SI calculus for implicit resolution.

- Intersection types, replacing compound types,
- Union types,
- Type lambdas, replacing encodings using structural types and type projection.
- Context Functions offering abstraction over given parameters.

Status: essential

These are essential core features of Scala 3. Without them, Scala 3 would be a completely different language, with different foundations.

Migration cost: none to low

Since these are additions, there's generally no migration cost for old code. An exception are intersection types which replace compound types with slightly cleaned-up semantics. But few programs would be affected by this change.

10.17.2 Simplifications

These constructs replace existing constructs with the aim of making the language safer and simpler to use, and to promote uniformity in code style.

- Trait parameters replace early initializers with a more generally useful construct.
- Given instances replace implicit objects and defs, focussing on intent over mechanism.
- Using Clauses replace implicit parameters, avoiding their ambiguities.
- Extension methods replace implicit classes with a clearer and simpler mechanism.
- Opaque type aliases replace most uses of value classes while guaranteeing absence of boxing.
- Top-level definitions replace package objects, dropping syntactic boilerplate.
- Export clauses provide a simple and general way to express aggregation, which can replace the previous facade pattern of package objects inheriting from classes.
- Vararg patterns now use the form : _* instead of @ _*, mirroring vararg expressions,
- Creator applications allow using simple function call syntax instead of new expressions. new expressions stay around as a fallback for the cases where creator applications cannot be used.

With the exception of early initializers and old-style vararg patterns, all superseded constructs continue to be available in Scala 3.0. The plan is to deprecate and phase them out later.

Value classes (superseded by opaque type aliases) are a special case. There are currently no deprecation plans for value classes, since we might want to bring them

back in a more general form if they are supported natively by the JVM as is planned by project Valhalla.

Status: bimodal: now or never / can delay

These are essential simplifications. If we decide to adopt them, we should do it for 3.0. Otherwise we are faced with the awkward situation that the Scala 3 documentation has to describe an old feature that will be replaced or superseded by a simpler one in the future.

On the other hand, we need to decide now only about the new features in this list. The decision to drop the superseded features can be delayed. Of course, adopting a new feature without deciding to drop the superseded feature will make the language larger.

Migration cost: moderate

For the next several versions, old features will remain available and deprecation and rewrite techniques can make any migration effort low and gradual.

10.17.3 Restrictions

These constructs are restricted to make the language safer.

- Implicit Conversions: there is only one way to define implicit conversions instead of many, and potentially surprising implicit conversions require a language import.
- Given Imports: implicits now require a special form of import, to make the import clearly visible.
- Type Projection: only classes can be used as prefix C of a type projection C#A. Type projection on abstract types is no longer supported since it is unsound.
- Multiversal Equality implements an "opt-in" scheme to rule out nonsensical comparisons with == and !=.
- infix makes method application syntax uniform across code bases.

Unrestricted implicit conversions continue to be available in Scala 3.0, but will be deprecated and removed later. Unrestricted versions of the other constructs in the list above are available only under -source 3.0-migration.

Status: now or never

These are essential restrictions. If we decide to adopt them, we should do it for 3.0. Otherwise we are faced with the awkward situation that the Scala 3 documentation has to describe a feature that will be restricted in the future.

Migration cost: low to high

- *low*: multiversal equality rules out code that is nonsensical, so any rewrites required by its adoption should be classified as bug fixes.
- *moderate*: Restrictions to implicits can be accommodated by straightforward rewriting.

• *high*: Unrestricted type projection cannot always rewritten directly since it is unsound in general.

10.17.4 Dropped Constructs

These constructs are proposed to be dropped without a new construct replacing them. The motivation for dropping these constructs is to simplify the language and its implementation.

- DelayedInit,
- Existential types,
- Procedure syntax,
- Class shadowing,
- XML literals,
- Symbol literals,
- Auto application,
- Weak conformance,
- Compound types,
- Auto tupling (implemented, but not merged).

The date when these constructs are dropped varies. The current status is:

- Not implemented at all:
 - DelayedInit, existential types, weak conformance.
- Supported under -source 3.0-migration:
 - procedure syntax, class shadowing, symbol literals, auto application, auto tupling in a restricted form.
- Supported in 3.0, to be deprecated and phased out later:
 - XML literals, compound types.

Status: mixed

Currently unimplemented features would require considerable implementation effort which would in most cases make the compiler more buggy and fragile and harder to understand. If we do not decide to drop them, they will probably show up as "not yet implemented" in the Scala 3.0 release.

Currently implemented features could stay around indefinitely. Updated docs may simply ignore them, in the expectation that they might go away eventually. So the decision about their removal can be delayed.

Migration cost: moderate to high

Dropped features require rewrites to avoid their use in programs. These rewrites can sometimes be automatic (e.g. for procedure syntax, symbol literals, auto application) and sometimes need to be manual (e.g. class shadowing, auto tupling). Sometimes the rewrites would have to be non-local, affecting use sites as well as definition sites (e.g., in the case of DelayedInit, unless we find a solution).

10.17.5 Changes

These constructs have undergone changes to make them more regular and useful.

- Structural Types: They now allow pluggable implementations, which greatly increases their usefulness. Some usage patterns are restricted compared to the status quo.
- Name-based pattern matching: The existing undocumented Scala 2 implementation has been codified in a slightly simplified form.
- Eta expansion is now performed universally also in the absence of an expected type. The postfix _ operator is thus made redundant. It will be deprecated and dropped after Scala 3.0.
- Implicit Resolution: The implicit resolution rules have been cleaned up to make them more useful and less surprising. Implicit scope is restricted to no longer include package prefixes.

Most aspects of old-style implicit resolution are still available under -source 3.0-migration. The other changes in this list are applied unconditionally.

Status: strongly advisable

The features have been implemented in their new form in Scala 3.0's compiler. They provide clear improvements in simplicity and functionality compared to the status quo. Going back would require significant implementation effort for a net loss of functionality.

Migration cost: low to high

Only a few programs should require changes, but some necessary changes might be non-local (as in the case of restrictions to implicit scope).

10.17.6 New Constructs

These are additions to the language that make it more powerful or pleasant to use.

- Enums provide concise syntax for enumerations and algebraic data types.
- Parameter Untupling avoids having to use case for tupled parameter destructuring.
- Dependent Function Types generalize dependent methods to dependent function values and types.
- Polymorphic Function Types generalize polymorphic methods to dependent function values and types. *Current status*: There is a proposal, and a prototype implementation, but the implementation has not been finalized or merged yet.
- Kind Polymorphism allows the definition of operators working equally on types and type constructors.

Status: mixed

Enums offer an essential simplification of fundamental use patterns, so they should be adopted for Scala 3.0. Auto-parameter tupling is a very small change that removes some awkwardness, so it might as well be adopted now. The other features constitute more specialized functionality which could be introduced in later versions. On the other hand, except for polymorphic function types they are all fully implemented, so if the Scala 3.0 spec does not include them, they might be still made available under a language flag.

Migration cost: none

Being new features, existing code migrates without changes. To be sure, sometimes it would be attractive to rewrite code to make use of the new features in order to increase clarity and conciseness.

10.17.7 Metaprogramming

The following constructs together aim to put metaprogramming in Scala on a new basis. So far, metaprogramming was achieved by a combination of macros and libraries such as Shapeless that were in turn based on some key macros. Current Scala 2 macro mechanisms are a thin veneer on top the current Scala 2 compiler, which makes them fragile and in many cases impossible to port to Scala 3.

It's worth noting that macros were never included in the Scala 2 language specification and were so far made available only under an -experimental flag. This has not prevented their widespread usage.

To enable porting most uses of macros, we are experimenting with the advanced language constructs listed below. These designs are more provisional than the rest of the proposed language constructs for Scala 3.0. There might still be some changes until the final release. Stabilizing the feature set needed for metaprogramming is our first priority.

- Match Types allow computation on types.
- Inline provides by itself a straightforward implementation of some simple macros and is at the same time an essential building block for the implementation of complex macros.
- Quotes and Splices provide a principled way to express macros and staging with a unified set of abstractions.
- Type class derivation provides an in-language implementation of the **Gen** macro in Shapeless and other foundational libraries. The new implementation is more robust, efficient and easier to use than the macro.
- Implicit by-name parameters provide a more robust in-language implementation of the Lazy macro in Shapeless.

Status: not yet settled

We know we need a practical replacement for current macros. The features listed above are very promising in that respect, but we need more complete implementations and more use cases to reach a final verdict.

Migration cost: very high

Existing macro libraries will have to be rewritten from the ground up. In many cases the rewritten libraries will turn out to be simpler and more robust than the

old ones, but that does not relieve one of the cost of the rewrites. It's currently unclear to what degree users of macro libraries will be affected. We aim to provide sufficient functionality so that core macros can be re-implemented fully, but given the vast feature set of the various macro extensions to Scala 2 it is difficult to arrive at a workable limitation of scope.

10.17.8 Changes to Type Checking and Inference

The Scala 3 compiler uses a new algorithm for type inference, which relies on a general subtype constraint solver. The new algorithm often works better than the old, but there are inevitably situations where the results of both algorithms differ, leading to errors diagnosed by Scala 3 for programs that the Scala 2 compiler accepts.

Status: essential

The new type-checking and inference algorithms are the essential core of the new compiler. They cannot be reverted without dropping the whole implementation of Scala 3.

Migration cost: high

Some existing programs will break and, given the complex nature of type inference, it will not always be clear what change caused the breakage and how to fix it.

In our experience, macros and changes in type and implicit argument inference together cause the large majority of problems encountered when porting existing code to Scala 3. The latter source of problems could be addressed systematically by a tool that added all inferred types and implicit arguments to a Scala 2 source code file. Most likely such a tool would be implemented as a Scala 2 compiler plugin. The resulting code would have a greatly increased likelihood to compile under Scala 3, but would often be bulky to the point of being unreadable. A second part of the rewriting tool should then selectively and iteratively remove type and implicit annotations that were synthesized by the first part as long as they compile under Scala 3. This second part could be implemented as a program that invokes the Scala 3 compiler scalac programmatically.

Several people have proposed such a tool for some time now. I believe it is time we find the will and the resources to actually implement it.