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.

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.

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 splices 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 splices, 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 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.

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.

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).

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).

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.

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.

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.

‹ Soft Ke...