SIP 18: Modularizing Language Features

Martin Odersky

14 April 2012

Change history

17 March 2012	First version
19 March 2012	Changed structuralTypes to reflectiveCalls. Introduced new flag for implicitConversions. Added Turing-completeness as a reason to control higherKinds.
20 March 2012	Changed flags to be types, made visibility requirements considerably more flexible.
29 March 2012	Changed flags to be implicit values of phantom types, to avoid shadowing problems.
14 April 2012	Adapted to actual implementation in trunk.

Motivation

The Scala language defines a range of powerful abstractions that enable the construction of high-level libraries and DSLs. As with every powerful feature, these have to be used with care and responsibility. With the increasing adoption of Scala in mainstream programming, we have seen rising demands for better guidance. Not every Scala programmer needs to make use of every specialized tool in Scala's arsenal to write libraries or DSLs. I therefore propose a scheme where some of the more advanced and contentious language features have to be enabled explicitly, typically using an import from a new `language` enumeration object (there are also alternative ways to enable a feature). The hope is that this will provide a good balance between the wish to provide the most powerful abstraction facilities possible and the wish to control of these features by making their usage more explicit.

The set of features so controlled is up to discussion. The current proposal includes two new constructs and five existing ones.

Status: The proposal has not yet been implemented, so should be seen as preliminary. It is to be expected that some details will change in the light of an implementation effort.

Specification

Let the `scala` package define a new object named `languageFeature` as follows:

```
object languageFeature {
  trait dynamics
  trait postfixOps
  trait reflectiveCalls
  trait implicitConversions
  trait higherKinds
  trait existentials
  object experimental {
    trait macros
  }
}
```

Furthermore, there is an object `language` in package `scala` that contains implicit feature values, where the name of the implicit value matches its feature type.

```
object language {
  import languageFeature._
  implicit val macros: macros = _
  implicit val dynamics: dynamics = _
  implicit val postfixOps: postfixOps = _
  implicit val reflectiveCalls: reflectiveCalls = _
  implicit val implicitConversions: implicitConversions = _
  implicit val higherKinds: higherKinds = _
  implicit val existentials: existentials = _
  object experimental {
    implicit val macros: macros = _
  }
}
```

The types in the `feature` object are called *feature flags*; they each control a set of features in the Scala language. To enable the a feature, an implicit value of the feature type must be available, which would typically happen with a named import from the `language` object. Examples:

```
  import language.experimental.macros
  import language.{reflectiveCalls, existentials}
```

The features controlled by flags are enabled in the whole scope where the import statement is effective.

More generally, a feature controlled by feature flag *X* is *enabled* in all program positions where an implicit argument of type `languageFeature.`*X* can be produced.

Feature flags can be imported wholesale, as in

```
 import language._
```

They can also be made visible through inheritance. So for instance to enable postfix operators in all code that inherits from `TestTrait`, you could define

```
      trait TestTrait {
        /** Allows postfix operators in test scripts */
        implicit val postfixOps = language.postFixOps
      }
```

The individual feature flags in `language` might vary in future releases. Furthermore, there is a transition period built in. Features that existed in Scala 2.9 and that are now controlled by a feature flag will trigger a warning when they are used where the flag is not enabled. In a future major Scala version that warning might be turned into an error.

I now present the set proposed for inclusion in 2.10, together with some arguments why the particular features are worth having but should be controlled.

The first two flags, `macros`, and `dynamics`, control new features of Scala. Using these features without the corresponding flag being enabled will lead to an error. Both of the features described are currently proposed in a SIP of which the outcome is not decided yet. Obviously, the discussion applies only if these SIPs are accepted.

[Update: SIP 17 was accepted, so `dynamics` is confirmed. SIP 16 was postponed, so `macros` is currently an experimental feature, and has to be imported from object `language.experimental`, or specified on the command line using `-language:experimental.macros`]

- `macros`. Where enabled, macro definitions are allowed. Macro implementations and macro applications are unaffected; they can be used anywhere. *Why introduce the feature?* Macros promise to make the language more regular, replacing ad-hoc language constructs with a general powerful abstraction capability that can express them. Macros are also a more disciplined and powerful replacement for compiler plugins. *Why control it?* For their very power, macros can lead to code that is hard to debug and understand.

- `dynamics`. Where enabled, direct or indirect subclasses of trait `scala.Dynamic` can be defined. Unless `dynamics` is enabled, a definition of a class, trait, or object that has `Dynamic` as a base trait is rejected. Dynamic member selection of existing subclasses of trait `Dynamic` are unaffected; they can be used anywhere. *Why introduce the feature?* To enable flexible DSLs and convenient interfacing with dynamic languages. *Why control it?* Dynamic member selection can undermine static checkability of programs. Furthermore, dynamic member selection often relies on reflection, which is not available on all platforms.

Another possible feature flag might control continuations, provided they are enabled by default in 2.10. This remains to be discussed.

The remaining flags control existing features of Scala. Using these features without enabling the corresponding flag will just lead to a warning for Scala 2.10, but might lead to an error for subsequent major versions of Scala.

- `postfixOps`. Only where enabled, postfix operator notation `(expr op)` will be allowed. *Why keep the feature?* Several DSLs written in Scala need the notation. *Why control it?* Postfix operators interact poorly with semicolon inference. Most programmers avoid them for this reason.

- `reflectiveCalls`. Only where enabled, accesses to members of structural types that need reflection are supported. Reminder: A structural type is a type of the form

  ```
  Parents { Decls }
  ```

  where `Decls` contains declarations of new members that do not override any member in `Parents`. To access one of these members a reflective call is needed. *Why keep the feature?* Structural types provide great flexibility because they avoid the need to define inheritance hierarchies a priori. Besides, their definition falls out quite naturally from Scala's concept of type refinement. *Why control it?* Reflection is not available on all platforms. Popular tools such as ProGuard have problems dealing with it. Even where reflection is available, reflective dispatch can lead to surprising performance degradations.

- `implicitConversions`. Only where enabled, definitions of implicit conversions are allowed. An implicit conversion is an implicit value of unary function type `A => B`, or an implicit method that has in its first parameter section a single, non-implicit parameter. Examples:

  ```
  implicit def stringToInt(s: String): Int = s.length
  implicit val conv = (s: String) => s.length
  implicit def listToX(xs: List[T])(implicit f: T => X): X = …
  ```

  Implicit values of other types are not affected, and neither are implicit classes as in SIP 13, if these are accepted. *Why keep the feature?* Implicit conversions are central to many aspects of Scala's core libraries. *Why control it?* Implicit conversions are known to cause many pitfalls if over-used. And we have noted a tendency to over-use them because they look very powerful and their effects seem to be easy to understand. Also, in most situations using implicit parameters leads to a better design than implicit conversions.

- `higherKinds.` Only where this flag is enabled, higher-kinded types can be written. *Why keep the feature?* Higher-kinded types enable the definition of very general abstractions such as functor, monad, or arrow. A significant set of advanced libraries relies on them. Higher-kinded types are also at the core of the scala-virtualized effort to produce high-performance parallel DSLs through staging. *Why control it?* Higher kinded types in Scala lead to a Turing-complete type system, where compiler termination is no longer guaranteed. They tend to be useful mostly for type-level computation and for a relatively narrow set of highly generic design patterns. The level of abstraction implied by these design patterns is often a barrier to understanding for newcomers to a Scala codebase. Some syntactic aspects of higher-kinded types are hard to understand for the uninitiated and type inference is less effective for them than for normal types. Because we are not completely happy with them yet, it is possible that some aspects of higher-kinded types will change in future versions of Scala. See my post "Scala - A Roadmap" to scala-language for details. So an explicit enabling also serves as a warning that code involving higher-kinded types might have to be revised in the future.

- `existentials.` Only where enabled, existential types that cannot be expressed as wildcard types can be written and are allowed in inferred types of values or return types of methods. Existential types with wildcard type syntax such as `List[_]`, or `Map[String, _]` are not affected. *Why keep the feature?* We need existential types to make sense of Java's wildcard types and raw types and the erased types of run-time values. *Why control it?* Having complex existential types in a code base usually makes application code very brittle, with a tendency to produce type errors with obscure error messages. Therefore, going overboard with existential types is generally perceived not to be a good idea. Also, complicated existential types might be no longer supported in a future simplification of the language. See my post “Scala - A Roadmap” to scala-language for details.

Alternative control using compiler options.

It is also possible to enable a language feature in all compiled files by specifying a command line option consisting of -language followed by a colon and the name of a feature flag. Example: To enable implicit conversions and higher-kinded types everywhere, compile with

```
scalac -language:implicitConversions -language:higherKinds
```

Finally, option `-language:_` would enable all feature flags, in correspondence to

```
import language._
```

This could be handy to start a REPL, say, where you want to experiment with all sorts of language features without further boilerplate. Just define an alias

```
alias repl = ‘scala -language:_’
```

and you are set.

Generalization of the scheme

The feature flags in language control certain language features that are typically very powerful but that can have hidden dangers or future compatibility implications. It’s intended generalize the scheme to other classes of features as well, but those classes should then not go into the language object itself but into some other object.

The first such extension is the subobject `experimental` of language which contains experimental features. Currently, the only feature flag in `language.experimental` is `macros`, but the scheme is intended to accommodate other experimental features in the future, obviating the need of the unfocused flag `-Xexperimental`.

Other possible generalizations, which are not part of this SIP are:

- a `future` object that controls features to be introduced in a future release

- a `deprecated` object that controls features no longer officially supported.
- a `plugin` object that controls loaded plugins