SIP: Self-cleaning macros

Eugene Burmako
Martin Odersky
Christopher Vogt
Stefan Zeiger
Adriaan Moors
scalamacros.org
https://github.com/scalamacros/kepler

09 March 2012

Motivation

Compile-time metaprogramming has been recognized as a valuable tool for enabling such programming techniques as:

- *Language virtualization* (overloading/overriding semantics of the original programming language to enable deep embedding of DSLs),
- *Program reification* (providing programs with means to inspect their own code),
- *Self-optimization* (self-application of domain-specific optimizations based on program reification),
- *Algorithmic program construction* (generation of code that is tedious to write with the abstractions supported by a programming language).

In this proposal we introduce *a macro system for Scala*. This facility allows programmers to write *macro defs:* functions that are transparently loaded by the compiler and executed during compilation. This realizes the notion of compile-time metaprogramming for Scala.

The proposed macro system holds the middle ground between hygienic and non-hygienic macros. The core macro expansion algorithm is non-hygienic. However, it is possible to write a macro def that brings hygiene to other macro defs. Hence, we call our macro system *self-cleaning*. This attribute goes along the lines of the fact that Scala macros are, in fact, cats.

Macro defs fit Scala's ecosystem by addressing problems that currently lack simple solutions. These include but are not limited to:

- *Advanced domain-specific languages design.* Macros are able to examine and restructure the ASTs of the program being compiled. As such, they can transparently override the semantics of Scala code with custom semantics of a DSL.
  For example, operations on collections can be transparently mapped to database queries.
- *Domain-specific optimizations.* With macro defs it becomes possible to apply domain-

specific knowledge to optimize programs during their compilation. We have optimized simple for-loops into close-to-metal while loops, removing the overhead of closures generated by Range.foreach, but retaining the same familiar syntax. Another use case for macros is elimination of boxing for certain use cases that involve primitive types and typeclasses.

- *Static checking of string literals and external DSLs.* This proposal synergizes with SIP: String interpolation and formatting. By implementing string processors as macro defs, the programmer can validate and process interpolated strings at compile time. This can be used to define custom literals (for example, binary numbers) or to deeply embed external domain-specific languages (for example, integrate HTML+CSS+JS into the namespace and type system of the main program). One of the potential improvements to Scala itself is moving XML literals from language level into the standard library, which is out of the scope of the current proposal, but might be implemented in future versions of Scala.

Macros are frequently compared with Virtualized Scala, a research project, done at EPFL and Stanford. At a glance, they look similar because both techniques enable language virtualization and algorithmic program construction. On the one hand, macros are more flexible than language virtualization because they do not rely on a universal lifting of types. On the other hand, language virtualization enforces a strict boundary between staged and unstaged expressions, which is often desired when implementing full-blown DSLs in Scala.

The intention of the present design is to have macros as a means to simplify things by offering uniform abstraction capabilities instead of ad-hoc extensions. At the same time, we are conscious that, like every abstraction, macros can be misused to obscure rather than clarify programs. Consequently, the design goals of the present proposal are first, minimality of the needed language abstractions, and second, expressiveness of the technique. Relatively less important for us was the convenience of writing macros, precisely because we anticipate that macros should be used only in relatively rare cases, and that they should be reserved to expert programmers who are familiar with language theory and compiler concepts. Put in a nutshell, we believe macros will be an attractive alternative to compiler plugins. They will not necessarily be a good alternative to the current abstraction capabilities such as higher-order functions.

Intuition

Here is a prototypical macro definition:

```
def m(x: T): R = macro implRef
```

At first glance macro definitions are equivalent to normal function definitions, except for their body, which starts with the conditional[1] keyword **macro** and is followed by a possibly qualified identifier that refers to a static macro implementation method.

If, during type-checking, the compiler encounters an application of the macro m(args), it will expand that application by invoking the corresponding macro implementation method, with the

[1] macro is currently considered to be a keyword only if the compiler flag -Xmacro is turned on. The name should be deprecated for identifiers in Scala 2.10, so that it can be used as a standard keyword in Scala 2.11.

abstract-syntax trees of the argument expressions args as arguments. The result of the macro implementation is another abstract syntax tree, which will be inlined at the call site and will be type-checked in turn.

Example 1. The following code snippet declares a macro definition assert that references a macro implementation Asserts.assertImpl.

```
def assert(cond: Boolean, msg: Any) = macro Asserts.assertImpl
```

A call assert(x < 10, “limit exceeded”) would then lead at compile time to an invocation

```
assertImpl(ctx)(<[ x < 10 ]>, <[ “limit exceeded” ]>)
```

where ctx is a *context* argument that contains information collected by the compiler at the call site, and the other two arguments are abstract syntax trees representing the two expressions x < 10 and “limit exceeded”.

Remark: In this document, <[expr]> denotes the abstract syntax tree that represents the expression expr. This notation has no counterpart in our proposed extension of the Scala language. In reality, the syntax trees would be constructed from the types in trait scala.reflect.api.Trees and the two expressions above would look like this:

```
Literal(Constant(“limit exceeded”))

Apply(
  Select(Ident(newTermName(“x”)), newTermName(“$less”),
  List(Literal(Constant(10))))
```

Here is a possible implementation of the assert macro:

```
import scala.reflect.makro.Context
object Asserts {
  def raise(msg: Any) = throw new AssertionError(msg)
  def assertImpl(c: Context)
        (cond: c.Expr[Boolean], msg: c.Expr[Any]) : c.Expr[Unit] =
    if (assertionsEnabled)
      <[ if (!cond) raise(msg) ]>
    else
      <[ () ]>
}
```

As the example shows, a macro implementation takes several parameter lists. First comes a single parameter, of type Context. This is followed by a list of parameters that have the same names as the macro definition parameters. But where the original macro parameter has type T, a macro implementation parameter has type c.Expr[T]. Expr[T] is a type defined in Context that wraps an abstract syntax tree of type T. The result type of the assertImpl macro implementation is again a wrapped tree, of type c.Expr[Unit].

Generic Macros

Macro definitions and macro implementations may both be generic. If a macro implementation has type parameters, actual type arguments must be given explicitly in the macro definition's body. Type parameters in an implementation may come with TypeTag context bounds. In that case the corresponding TypeTags describing the actual type arguments instantiated at the application site will be passed along when the macro is expanded.

Example 2 The following code snippet declares a macro definition Queryable.map that references a macro implementation QImpl.map:

```
class Queryable[T] {
  def map[U](p: T => U): Queryable[U] = macro QImpl.map[T, U]
}

object QImpl {
  def map[T: c.TypeTag, U: c.TypeTag]
         (c: Context)
         (p: c.Expr[T => U]): c.Expr[Queryable[U]] = …
}
```

Now consider a value q of type Queryable[String] and a macro call

```
      q.map[Int](s => s.length)
```

The call is expanded to the following reflective macro invocation

```
      QImpl.map(ctx)(<[ s => s.length ]>)
        (implicitly[TypeTag[String]], implicitly[TypeTag[Int]])
```

Standard Types

The previous discussion has outlined essentially all that's needed in the Scala language to support macros. The next part of the proposal deals with the standard types that are used in the signatures of macro implementations.

Macro Contexts

The context parameter of a macro implementation fulfils two roles. First, it provides a common frame of reference for the parameters that follow it. Scala reflection defines concepts such as abstract syntax trees, types, or symbols as members of a *mirror*. All mirrors are instances of the class scala.reflect.api.Universe, which contains inherited type members Tree (for abstract syntax trees), Symbol (for defined entities), Type (for their types), and several others more. Each instance of a Scala compiler is itself a reflection mirror. Macros get passed this mirror as part of the context parameter so that they can operate on trees and other data structures that are defined in the Scala compiler.

Contexts also provide some info regarding the environment of the macro call site. In particular, they record (in the field prefix) the abstract syntax tree that preceded the macro if the macro method is a member of some class or object. For example, a macro call x.y.m(arg) results in a context where the prefix field points to the abstract syntax tree <[x.y]>. Contexts may expose information such as the name of the compilation unit, or the line number where the macro was called. They may also offer selected compiler methods that macros can invoke. In fact, the specification leaves the door open for a hierarchy of context types that expose varying levels of capabilities. A macro implementation declares the level of power it needs by the type of its macro parameter. Details will be specified in the full definition of class scala.reflect.makro.Context and its subclasses.

Here is a minimal version of class scala.reflect.makro.Context.

```
abstract class Context {

  /** The mirror that represents the compile-time universe */
  val mirror: api.Universe

  /** Aliases of mirror types */
  type Symbol = mirror.Symbol
  type Type = mirror.Type
  type Name = mirror.Name
  type Tree = mirror.Tree
  type Expr[T] = mirror.Expr[T]
  type TypeTag[T] = mirror.TypeTag[T]

  /** Creator/extractor objects for Expr and TypeTag values */
  val TypeTag = mirror.TypeTag
  val Expr = mirror.Expr

  /** The type of the prefix tree from which the macro is selected */
  type PrefixType

  /** The prefix tree from which the macro is selected */
  val prefix: Expr[PrefixType]

  /** Reification yielding an Expr bound to the current mirror */
  def reify[T](expr: T): Expr[T]
```

```
 … // information about the call site and other operations
}
```

Type tags

A value of type TypeTag[T] encapsulates a representation of type T. It is supposed to replace the current concept of a Manifest. TypeTags are much better integrated with reflection than manifests are, and are consequently much simpler. Type tags are organized in a hierarchy of three classes: ClassTag, TypeTag and GroundTypeTag. These are defined as members of scala.reflect.api.Universe as follows:

```
  case class ClassTag[T](erasure: java.lang.Class[_]) {
    def tpe: Type = classToType(erasure)
  }

  case class TypeTag[T](tpe: Type) {
    def erasure: java.lang.Class[_] = typeToClass(tpe)
  }

  class GroundTypeTag[T](tpe: Type) extends TypeTag[T](tpe)
```

A ClassTag value wraps a Java class, which can be accessed via the erasure method. A TypeTag value wraps a full Scala type in its tpe field. A GroundTypeTag value is a type tag that is guaranteed not to contain any references to type parameters or abstract types.

Implicit in the contract for all Tag classes is that the reified type tpe represents the type parameter T. Tags are typically created by the compiler, which makes sure that this contract is kept. The creation rules are as follows:

1. If an implicit value of type u.ClassTag[T] is required, and T is a class type, the compiler will make one up on demand. The implicitly created value contains in its erasure field the class of type T.
2. If an implicit value of type u.TypeTag[T] is required, the compiler will make one up on demand. The implicitly created value contains in its tpe field a value of type u.Type that is a reflective representation of T. In that value, any occurrences of type parameters or abstract types U which come themselves with a TypeTag are represented by the type referenced by that TypeTag.
3. If an implicit value of type u.GroundTypeTag[T] is required, the compiler will make one up on demand following the same procedure as for TypeTags. However, if the resulting type still contains references to type parameters or abstract types, a static error results.

An example that illustrates the TypeTag embedding described in item 2, consider the following function:

```
        import reflect.mirror._
        def f[T: TypeTag, U] = {
          type L = T => U
          implicitly[TypeTag[L]]
        }
```

Then a call of f[String, Int] will yield a result of the form

```
        TypeTag(<[ String => U ]>).
```

Note that T has been replaced by String, because it comes with a TypeTag in f, whereas U was left as a type parameter.

TypeTags correspond loosely to Manifests. More precisely, the previous notion of a ClassManifest corresponds to a ClassTag, the previous notion of a Manifest corresponds to GroundTypeTag, whereas TypeTag is approximated by the previous notion of OptManifest. It is planned that the previous names Manifest, ClassManifest and OptManifest are kept around as deprecated aliases of the corresponding Tag classes.

Expression trees

An expression tree of type Expr[T] encapsulates an abstract syntax tree of type T and its type. Here's the definition of Expr as a member of scala.reflect.api.Universe:

```
  case class Expr[T](tree: Tree) {
    def eval: T = …
    lazy val value: T = eval
  }
```

Implicit in the contract for Expr is that the type of the reified tree conforms to the type parameter T. Expr values are typically created by the compiler, which makes sure that this contract is kept.

Note that the method eval which when called on a value of type Expr[T] will yield a result of type T. The eval method and the value value play a special role in tree splicing (see below).

What about Hygiene?

The macro scheme described so far has the advantage that it is minimal, but also suffers from two inconveniences: Tree construction is cumbersome and hygiene is not guaranteed. Consider again a fragment of the body of assertImpl in Example 1:

```
<[ if (!cond) raise(msg) ]>
```

To actually produce the abstract syntax tree representing that expression one might write something like that:

```
c.Expr(
  If(Select(cond, newTermName("unary_$bang")),
     Apply(Ident(newTermName("raise")), List(msg)),
     Literal(Constant(()))))
```

Cumbersome enough as this is, it is also wrong. Remember that the tree produced from a macro will be inlined and type-checked at the macro call site. But that means that the identifier raise will be typechecked at a point where it is most likely not visible, or in the worst case they might refer to something else. In the macro literature, this insensitivity to bindings is called "unhygienic". In the case of assertImpl, the problems can be avoided by generating instead of an identifier a fully qualified selection

```
Select(Asserts, newTermName("raise"))
```

(to be 100% sure, one would need to select the full path starting with the root package). But that makes the tree construction even more cumbersome and is very fragile because it is easily forgotten.

However, it turns out that macros themselves can be used to solve both these problems. A corner-stone of the technique is a macro called reify that produces its tree one stage later.

Reify

The reify macro plays a crucial role in the proposed macro system. It's definition as a member of Context is:

```
def reify[T](expr: T): Expr[T] = macro ...
```

Reify accepts a single parameter expr, which can be any well-typed Scala expression, and creates a tree that, when compiled and evaluated, will recreate the original tree expr. So reify is like time-travel: Trees get re-constituted at a later stage. If reify is called from normal compiled code, its effect is that the abstract syntax tree passed to it will be recreated at run time. Consequently, if reify is called from a macro implementation, its effect is that the abstract syntax tree passed to it will be recreated at macro-expansion time (which corresponds to run time for macros). This gives us a convenient way to create syntax trees from Scala code: just pass the Scala code to reify, and the result will be a syntax tree that represents that very same code.

What's more, reify packages the result expression tree with the types and values of all free references that occur in it. This means in effect that all free references in reify's result are already resolved, so that re-typechecking the tree is insensitive to its environment. All identifiers referred to from an expression passed to reify are bound at the definition site, and not re-bound at the call site. As a consequence, macros that generate trees only by the means of passing expressions to reify are hygienic.

So in that sense, the proposed Scala macros are self-cleaning. Their basic form is minimal and unhygienic, but that simple form is expressive enough to formulate a reify macro, which can be used in turn to make tree construction in macros concise and hygienic.

Example 3:

Here is an implemention of the assert macro using reify.

```
import scala.reflect.makro.Context
object Asserts {
  def raise(msg: Any) = throw new AssertionError(msg)
  def assertImpl(c: Context)
        (cond: c.Expr[Boolean], msg: c.Expr[Any]) : c.Expr[Unit] =
    if (assertionsEnabled)
      c.reify(if (!cond.eval) raise(msg.eval))
    else
      c.reify(())
}
```

Note the close correspondence with the meta-notation of Example 1.

Splicing

Reify and eval are inverses of each other. Reify takes an expression and produces a tree that, when evaluated with eval, yields the same result as the original expression. This is also expressed by their types. reify goes from T to Expr[T], and eval goes from Expr[T] back to T. The reify macro takes advantage of this relationship by shortcircuiting embedded calls to eval:

```
reify(expr.eval)          translates to    expr
```

This principle is seen in action in Example 3 above. There, the contents of the parameters cond and msg are “spliced” into the body of the reify.

Along with eval, value also gets special treatment by reify.

```
reify(expr.value) also translates to  expr
```

Similar to eval, the value method also makes reify splice its tree into the result. The difference appears when the same expression gets spliced into multiple places inside the same reify block. With eval, reify will always insert a copy of the corresponding tree (potentially duplicating side-effects), whereas value will splice itself into a temporary variable that will be referred by its usages.

The notion of splicing also manifests itself when reify refers to a type that has a TypeTag associated with it. In that case instead of reproducing the type's internal structure as usual, reify inserts a reference to the type tag into its result.

reify(T) translates to TypeTag[T].tpe

Tagging a type can be done either automatically, by writing a c.TypeTag context bound on a type parameter of a macro implementation, or manually, by introducing an implicit TypeTag value into the scope visible by reify.

Specification

Syntax

Macro definitions extend the grammar described in "4.6 Function Declarations and Definitions" of SLS 2.9 with the following syntax:

```
Def ::= 'def' FunDef | 'def' MacroDef
FunDef ::= FunSig [':' Type] '=' Expr
MacroDef ::= FunSig [':' Type] '=' 'macro' QualId [TypeArgs]
```

Note We propose to make "macro" a new keyword. This does not break backward compatibility, because macro definitions can only be compiled under the -Xmacros compiler flag. For the next version of Scala we suggest prohibiting identifiers named "macro" in programs compiled with -Xmacros and deprecating such identifiers in programs compiled normally.

Note also that the syntax does not contain clauses for macro declarations. In other words, macros cannot be abstract.

Typing Rules

The signatures of macro definitions and function definitions are identical. Macro definitions can be declared in all contexts legal for declaring a function/method, with the following restrictions on overriding.

A macro may override another macro or a concrete method (in both cases the override modifier is required as usual), but it may not directly implement an abstract method that is not also implemented by an inherited concrete method. However, macro selection always depends on the static type of the receiver object, not the dynamic type. Macros may themselves be overridden only by macros, not by other methods.

Example 4: Given:

```
    class C { def m() = 1 }
    class D extends C { override def m() = macro mI }
```

In that situation, the sequence

```
    val d = new D; d.m()
```

will lead to the expansion of macro m. However, the sequence

```
    val c: C = d; c.m()
```

will lead to the invocation of method m in C, even though the dynamic receiver type is D.

The body of a macro definition consists of a qualified identifier that references a static method, possibly followed by type arguments. Here, a method definition is *static* if it is a member of a static object. An object definition is static if it is a toplevel definition or if it is a member of a static object.

The macro implementation referenced in the body must be compatible with the macro definition, as expressed by the following definition.

Definition The macro implementation reference MI or MI[T1, …, Tn] is *compatible* with the macro definition MD if the following holds.

1. MI has n type parameters (where n = 0 in the case of a parameterless reference MI), and the type arguments [T1, …, Tn] are legal instantiations of these type parameters. Any context bounds of these type parameters are assumed in the following to be already expanded to implicit parameters.
2. The first parameter list of MI contains a single parameter of a compiler-supported subtype of scala.reflect.makro.Context. Scala implementations may support subclasses of Context in this position, which may reveal more detail than the standard scala.reflect.makro.Context. Also permitted is a refinement on the PrefixType member of class Context. The permissible refinement is { type PrefixType = σU }, provided MD is a member of a class or trait with self type U. Let c be the name of the context parameter.
3. The following parameter list(s) of MI correspond one-by-one to the parameter list(s) of MD. Corresponding lists have the same number of parameters. Corresponding parameters must have the same name, and where MD’s parameter has type T, the corresponding MI parameter must have a type U such that c.Expr[σT] conforms to U.
4. Afterwards, MI may optionally have one extra implicit parameter list. All parameters in

that list must be of form c.TypeTag[T], where T is a type parameter of MI. (It follows that the type parameters of MI may have no context- or view-bounds except TypeTag bounds)

5. If the (declared or assumed) result type of MD is T then the result type of MI must conform to c.Expr[σT].

Here, σ is a substitution that maps every type parameter of MI to the corresponding type argument in the body of the macro call, and that maps every value parameter x of MD to x'.value, where x' is the corresponding value parameter in MI (which is also named x). If MD is defined in a class or trait C, σ also maps every occurrence of C.this to c.prefix.value, where c is the first context parameter of MI.

Note: σ allows to implement macro definitions with dependent method types. For instance a macro definition

```
def selectT(x: C): x.Tpe = macro selectTImpl
```

would correspond to the macro implementation

```
def selectTImpl(c: Context)(x: c.Expr[T]): c.Expr[x.value.Tpe]
```

Without the mediating power of σ, the implementation result type c.Expr[x.value.Tpe] could not be recognized as compatible with the definition result type x.Tpe.

The result type of MD may be omitted, in which case it is inferred from the result type of MI. If MI's result type has a base type of the form c.Expr[T], then MD's result type is taken to be T. Otherwise, MD's result type is taken to be Any.

Macro expansion

Unlike normal functions, macros must always be fully applied; eta expansion is not available for macros (this includes method value syntax and partial applications). Macro applications are processed statically and, hence, cannot be manipulated as higher-order functions.

Also, in the present proposal, macro applications cannot have named or default arguments (we might be able to lift this restriction in the future).

Let   q.m[Ts](args1)...(argsn)   be a fully applied application of the macro m to the type arguments Ts and the value argument args (either arguments may be missing). The expression is typechecked first in exactly the same way as a normal method application, followed by a macro expansion step. The expansion step performs a reflective invocation
of the implementation method of macro m, with the following arguments in the order they are given:

1. The first argument list consists of an value of type scala.reflect.makro.Context, which contains the qualifier q as prefix field.
2. The following argument lists correspond one-by-one to args1 .. argsn. Each value argument is passed as-is to the macro implementation, wrapped in an expression value

of type Expr.

3. If the macro implementation has implicit TypeTag parameters, this is followed by a final list which passes for every parameter of type TypeTag[Vi] a value of type TypeTag[Ti]. Here, Vi is a type parameter of MI and Ti is the corresponding type argument in the macro implementation reference.

It is an error if the macro implementation does not exist or is not accessible at the time the macro is expanded (this means that macro implementations need to be compiled before macro uses; whereas no such ordering requirement exists for macro definitions).

If the macro invocation returns with an Expr value, the macro application is replaced by the abstract syntax tree contained in that value. Otherwise, a static error results. (Scala implementations may impose a timeout on macro invocations in order to guard against infinitely looping macro implementations). If an Expr value is returned, its tree is then type-checked in turn and is expected to conform to the result type of the original macro application.

Note. If the result of macro expansion itself contains macro applications, then the algorithm will perform recursive expansion upon them, because expansion is integrated into typechecking. This means that macro expansion is recursive.

Implementation Details

This section covers implementation details of the macro system described in this proposal, including the details of loading macro implementations, compiler API exposed to macros and debugging.

- Macro *definitions* can only be compiled when the -Xmacros compiler flag is passed to the compiler. However, macro implementations and macro invocations do not need any special flags.

- When loading a macro implementation, the Scala compiler uses the library classpath (i.e. the normal classpath of the compilation). This means that to use a macro, one only needs to add its implementation classes to the classpath.

- As mentioned above, macro implementations must be compiled in a compilation run that is different from the one that performs macro expansion. REPL makes this a non-issue, because every command to a REPL gets compiled in a separate compilation run. Hence it is possible to define and use macros in the same REPL session.

- To debug a macro one needs to run Scala compiler under the debugger (with the entry point being scala.tools.nsc.Main) and it becomes possible to put breakpoints and watch local variables in macro bodies. Another option is tracing macro compilation and expansion by using -Ymacro-debug. If along with -Ymacro-debug, the -Ymacro-copypaste option is enabled as well, Scala compiler will also dump expansions in the form that can be copy/pasted into a standalone program or a REPL.

- Debugging programs generated by macros is hard, because there’s no source code that one can put breakpoints on. This might be improved in future versions of Scala.

- In an IDE, one need to keep the original macro application around even after macro expansion. Otherwise, macro arguments could no longer be found and semantically analyzed.