Scala Macros, a Technical Report

Eugene Burmako, Martin Odersky

École Polytechnique Fédérale de Lausanne (EPFL)
`first.last@epfl.ch`

Abstract. Metaprogramming is a powerful technique of software development, which allows to automate program generation. Applications of metaprogramming range from improving expressiveness of a programming language via deep embedding of domain-specific languages to boosting performance of produced code by providing programmer with fine-grained control over compilation. In this report we introduce macros, facility that enables compile-time metaprogramming in the Scala programming language.

Keywords: Compile-time Metaprogramming, Macros, Multi-stage Programming, Language Virtualization

1 Introduction

As its name suggests, Scala (which stands for "scalable language" [1]) has been built from the ground up with extensibility in mind. Such features as abstract type members, explicit selftypes and modular mixin composition enable the programmer to compose programs as systems of reusable components [2].

The symbiosis of language features employed by Scala allows the code written in it to reach impressive levels of modularity [3], however there is still room for improvement. For example, the semantic gap between high-level abstractions and the runtime model of Java Virtual Machine brings performance issues that become apparent in high-performance scenarios [5]. Another example is state of the art in data access techniques. Recently established standards in this domain [4] cannot be readily expressed in Scala, which represents a disadvantage for enterprise software development.

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) [6], *program reification* (providing programs with means to inspect their own code) [8, 10], *self-optimization* (self-application of domain-specific optimizations based on program reification) [11, 12], *algorithmic program construction* (generation of code that is tedious to write with the abstractions supported by a programming language) [7, 8].

Our research introduces new concepts to Scala programming languages enabling metaprogramming techniques that address modern development challenges in an approachable and structured way [9].

2 Intuition

Here is a prototypical macro definition in our macro system:

```
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 keyword `macro` and is followed by a possibly qualified identifier that refers to a 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 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(c)(<[ x < 10 ]>, <[ "limit exceeded" ]>)
```

where `c` is a context argument that contains information collected by the compiler at the call site (receiver of the macro invocation, symbol tables for enclosing lexical scopes, etc.), and the other two arguments are abstract syntax trees representing the two expressions `x < 10` and `"limit exceeded"`.

In this document `<[ expr ]>` denotes the abstract syntax tree that represents the expression `expr`, but this notation has no counterpart in our extension of the Scala language. The canonical way to construct abstract syntax trees is to use the types in the compiler library, which for the two expressions above looks like this:

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

Literal(Constant("limit exceeded"))
```

The core of our macro system is described in sections 3 through 6 and is inspired by the notions from LISP [13], Scheme [14] and Nemerle [8]. Sections 7 through 9 describe a peculiar feature of Scala macros that makes use of staging to bootstrap macros into a hygienic and quasiquoting metaprogramming system. Subsequent sections conclude the report.

3 Baseline

Let us examine a possible implementation of the assert macro mentioned in *Example 1* to explore the foundations of Scala macros:

```
object Asserts {
  def assertionsEnabled = ...
  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 listing 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]`.

Parameters of a macro implementation are dependently typed, being a part of a dependent method type [15]. Such type annotations statically ensure that artifacts passed into a macro belong to the context that services a macro expansion. This type-checking facility is important from a practical standpoint, as it prevents accidental mix-up of compilation stages. For example, without dependent typing it would be possible to inadvertently refer to runtime trees and types (obtained from a reflection context) in a compile-time macro (that uses the compiler context).

The macro being discussed is static, in a sense that it has a statically known receiver (such receivers are called “objects” in Scala parlance). It is possible, however, to define instance macros and use them in a prefix fashion, analogously to instance methods, e.g. `receiver.a_macro(args)`. In that case, abstract syntax tree corresponding to `receiver` is passed to the macro implementation in `Context`.

4 Expression Trees

An expression tree of type `Expr[T]` encapsulates an abstract syntax tree of type `T` together with its type. Heres the definition of `Expr` as a member of the compiler library exposed to macro implementations:

```
 case class Expr[T: TypeTag](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` (which is also reified by the virtue of the `TypeTag` context bound, as described in subsequent sections). `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 as described in subsequent sections.

5 Polymorphic Macros

Macro definitions and macro implementations may both be polymorphic.

Type parameters in an implementation may come with `TypeTag` context bounds [16]. 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 code below declares a polymorphic macro definition `Queryable.map` that references a polymorphic 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]] = ...
}
```

As outlined in [16], context bounds provide a concise notation for declaring implicit parameter sections that captures suitable type class instances from lexical scope. For example, method `QImpl.map` is desugared into the following form:

```
object QImpl {
  def map[T, U]
         (c: Context)
         (p: c.Expr[T => U])
         (implicit evidence$1: c.TypeTag[T],
          implicit evidence$2: c.TypeTag[U])
         : c.Expr[Queryable[U]] = ...
}
```

Now consider a value `q` of type `Queryable[String]` and the following macro call (the explicit type argument `[Int]` can be omitted, in which case it will be inferred by the compiler):

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

The call is expanded to the following macro invocation:

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

The `implicitly` function is used to summon suitable (i.e. marked as implicit and having a conformant type signature) type tags from the lexical scope of the call site. Shortly put, implicit search starts with the innermost enclosing scope and proceeds from the inside out (details about the implicit resolution algorithm may be found in [17]).

Of course, macro runtime does not expect the programmer to know about macro contexts, to create the type tags manually and to put them into local variables visible from macro call sites. In a common case when type tags are not declared explicitly, implicit search will fall back to the outermost scope, declared in the standard library. This outermost scope hosts implicit macros that are capable of materializing type tags for arbitrary types.

6 Type Tags

A value of type `TypeTag[T]` encapsulates a representation of type `T`. A `TypeTag` value simply wraps a Scala type, while a `ConcreteTypeTag` value is a type tag that is guaranteed not to contain any references to type parameters or abstract types.

```
case class TypeTag[T](tpe: Type) { ... }
class ConcreteTypeTag[T](tpe: Type) extends TypeTag[T](tpe)
```

Implicit in the contract for all `Tag` classes is that the reified type 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 `TypeTag[T]` is required, the compiler will summon it from the enclosing lexical scope or make one up on demand using the implicit search algorithm described in the previous section.

2) The implicitly created value contains a value of type `Type` that is a reified representation of `T`. In that value, any occurrences of type parameters or abstract types `U` which come themselves with a `TypeTag` are represented by that `TypeTag`. This is called type splicing.

3) If an implicit value of type `ConcreteTypeTag[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.

As an example that illustrates type splicing, consider the following function:

```
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.

Type splicing plays an important role in the design of the metaprogramming system, because Scala uses the erase-types model to compile polymorphic types [18]. In this model, when the program is compiled down to executable form, type arguments of the invocations are removed, and type parameters are replaced by their upper bounds. With implicit parameters it becomes possible to capture type arguments during the compile-time to retain them at runtime [3], and type splicing scales this technique to complex types.

7 Quasiquoting and 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 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. 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 type-checked 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 non-hygienic [19, 8].

In the case of `assertImpl`, the problems can be avoided by generating instead of an identifier a fully qualified selection

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

(to be completely 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.

8 Reify

The reify macro plays a crucial role in the proposed macro system. Its 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 a convenient way to create syntax trees from Scala code: pass the Scala code to `reify`, and the result will be a syntax tree that represents that very same code.

Moreover, `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 the 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, 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 implementation of the assert macro using `reify`.

```
object Asserts {
  def assertionsEnabled = ...
  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*.

9 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 short-circuiting embedded calls to `eval` during the process that we call tree splicing (compare this with type splicing described above):

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

This principle is seen in action in *Example 3*. 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:

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

Similar to `eval`, the `value` value 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 types internal structure as usual, `reify` inserts a reference to the type tag into its result.

```
reify(expr: T) translates to expr typed as TypeTag[T].tpe
```

Tagging a type can be done either automatically, by writing a `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`.

Note the close resemblance of type splicing in `reify` and type splicing during `TypeTag` generation. In fact, here we are talking about the same algorithm. When producing a `TypeTag` for a type, corresponding implicit macros call `reify` (which, in turn, calls `TypeTag` generators to resolve potential splices using the implicit search algorithm).

10 Related Work

The history of compile-time metaprogramming dates back to the times of LISP [13], which was introduced in 1950s. Since then a fair amount of languages: statically typed [7] and dynamically typed [20], minimalistic [14] and having rich syntax [8] - have adopted macros. Our research builds on this notion of compile-time program transformation.

Hygiene is an important idea brought up in Scheme. The problem of inadvertent name clashes between the application code and generated macro expansions has been well-known in the Lisp community. Kohlbecker et al. [19] have solved this problem by embedding the knowledge of declaration sites into symbols that represent values and declarations in the program, making macro expansions hygienic.

We acknowledge this problem, but our means of achieving hygiene do not require changes to the type-checking algorithm. By making use of `reify`, a staging macro, we statically ensure that cross-stage bindings do not occur. Similar approach has been used in MacroML [21], which implements a macro system in a staged language MetaML [22]. Our arrival at this conflux of ideas happened the other way around - we built a staged system with macros.

As a language with syntax, Scala has to work hard to achieve homoiconicity. The incovenience of manipulating abstract syntax trees in their raw form is a well-known problem, and it affects rich languages to a greater extent than it affects minimalistic ones. Traditional solution to this problem is a quasiquoting DSL that lets the programmer encode ASTs in a WYSIWYG manner [26, 8, 27].

Our answer to this challenge is the same staging macro `reify` that we use to achieve hygiene. Code passed to `reify` becomes an AST one stage later, which provides a quasiquoting facility without the need to introduce a special domain-specific language.

Finally, as a language virtualization platform, Scala macros are conceptually close to Scala-Virtualized [23] which virtualizes base language constructs (e.g. control flow) and even data structures [24]. However, our approach to virtualization is different. Scala macros expose general-purpose Scala trees and types and provide low-level manipulation facilities, whereas Scala-Virtualized is good for embedded DSLs, in particular when the DSL expression trees do not exactly correspond to Scala trees [25].

11 Conclusions

We have presented a minimalistic macro system for Scala, a language with rich syntax and static types. This macro system builds up a metaprogramming facility on a single concept - compile-time AST transformer functions.

Other metaprogramming facilities usually include additional concepts of quasiquoting and hygiene to make themselves suitable for practical use. We have shown, however, that it is possible to implement both on top of our minimalistic core.

Acknowledgements

The authors would like to thank Vladislav Chistyakov, Jan Christopher Vogt, Stefan Zeiger, Adriaan Moors and the entire Scala community for insightful discussions and helpful comments.

References

1. Odersky, M., Spoon L., and Venners B., *Programming in Scala, Second Edition.* Artima Press, 2010.
2. Odersky, M., and Zenger M., *Scalable Component Abstractions.* ACM Sigplan Notices, 2005.
3. Odersky, M., and Moors, A., *Fighting Bit Rot with Types (Experience Report: Scala Collections).* Theoretical Computer Science, 2009.
4. Box, D., and Hejlsberg, A., *LINQ: .NET Language-Integrated Query*, Retrieved from http://msdn.microsoft.com/en-us/library/bb308959.aspx, 2007.
5. Dragos I., *Optimizing Higher-Order Functions in Scala*, Third International Workshop on Implementation Compilation Optimization of ObjectOriented Languages Programs and Systems, 2008.

6. McCool, M. D., Qin, Z., and Popa, T. S., *Shader metaprogramming*, Proceedings of the ACM SIGGRAPHEUROGRAPHICS conference on Graphics hardware, 2002.
7. Sheard, T., and Peyton Jones, S., *Template Meta-programming for Haskell*, Haskell Workshop, 2002.
8. Skalski K., *Syntax-extending and type-reflecting macros in an object-oriented language*, Master Thesis, 2005.
9. Scala Macros, *Use cases*, Retrieved from http://scalamacros.org/usecases.html, 2012.
10. Attardi, G., and Cisternino, A., *Reflection support by means of template metaprogramming*, Time, 2001.
11. Seefried, S., Chakravarty, M., and Keller, G., *Optimising Embedded DSLs using Template Haskell*. Generative Programming and Component Engineering, 2004.
12. Cross, J., and Schmidt, D., *Meta-Programming Techniques for Distributed Real-time and Embedded Systems*, 7th IEEE Workshop on Object-oriented Real-time Dependable Systems, 2002.
13. Steele, G., *Common LISP. The Language. Second Edition*, Digital Press, 1990.
14. *The Revised [6] Report on the Algorithmic Language Scheme*, Journal of Functional Programming, volume 19, issue S1, 2010.
15. Odersky, M., Cremet, V., Rckl, C., and Zenger M., *A Nominal Theory of Objects with Dependent Types*, 17th European Conference on Object-Oriented Programming, 2003.
16. Oliveira, B., Moors, A., and Odersky, M., *Type classes as objects and implicits*, 25th Conference on Object-Oriented Programming, Systems, Languages & Applications, 2010.
17. Odersky, M., *The Scala Language Specification, Version 2.9*, 2011.
18. Schinz, M., *Compiling Scala for the Java Virtual Machine*, PhD thesis, 2005.
19. Kohlbecker, E., Friedman, D., Felleisen, M., and Duba, B., *Hygienic macro expansion*, Symposium on LISP and Functional Programming, 1986.
20. Rahien, A., *DSLs in Boo: Domain-Specific Languages in .NET*, Manning Publications Co., 2010.
21. Ganz, S., Sabry, A., and Taha, W., *Macros as Multi-Stage Computations: Type-Safe, Generative, Binding Macros in MacroML*, International Conference on Functional Programming, 2001.
22. Taha, W., and Sheard, T., *MetaML: Multi-Stage Programming with Explicit Annotations*, 1999.
23. Moors, A., Rompf, T., Haller, P., and Odersky, M., *Scala-Virtualized*, Partial Evaluation and Program Manipulation, 2012.
24. Slesarenko A, *Lightweight Polytypic Staging: a new approach to Nested Data Parallelism in Scala*, Scala Days, 2012.
25. Rompf, T., and Odersky, M., *Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs*, 2010.
26. Bawden, A., *Quasiquotation in Lisp*, Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and SemanticsBased Program Manipulation, 1999.
27. Mainland, G., *Why it's Nice to be Quoted: Quasiquoting for Haskell*, Applied Sciences, 2007.