Project Kepler: Compile-Time Metaprogramming for Scala

Eugene Burmako

École Polytechnique Fédérale de Lausanne

Semester project report, Fall 2011-2012

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 [1] to boosting performance of produced code by providing programmer with fine-grained control over compilation [2]. In this report we introduce macros and quasiquotations, facilities that enable compile-time metaprogramming in Scala programming language.

1 Introduction

As its name suggests, Scala (which stands for "scalable language" [3]) 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 [4].

The symbiosis of language features employed by Scala allows the code written in it to reach impressive levels of modularity [5], however there is still some 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 [7, 8]. Another example is state of the art in data access techniques. Recently established standards in this domain [6] 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) [9], *program reification* (providing programs with means to inspect their own code) [11, 13], *self-optimization* (self-application of domain-specific optimizations based on program reification) [2, 14], *algorithmic program construction* (generation of code that is tedious to write with the abstractions supported by a programming language) [10, 11].

Our research introduces new concepts to Scala programming languages (several flavors of macros along with quasiquotations) enabling metaprogramming techniques that address modern development challenges in an approachable and structured way [12].

2 Proposed extensions to Scala

We propose to enrich Scala with three flavors of type-safe compile-time macros: macro defs, macro types and macro annotations. These kinds of macros can be used wherever their vanilla analogues are used, but they are treated differently by the compiler, providing novel ways to analyze and generate code [12].

Conceptually, macros [10, 11, 15] are very close to compiler plugins [16]. Similarly to compiler plugins, macros run inside the compiler and operate on code as data in the form of abstract syntax trees. However, macros are more lightweight (don't require wrapping in boilerplate) and less universal (can inject into the compiler only in predefined extension points).

Along with macros, our proposal suggests to extend Scala with quasiquotations [11, 17, 18] - special kinds of string literals that are parsed by Scala compiler and get transformed into abstract syntax trees. Quasiquotes can also be composed and decomposed in a simple fashion, which simplifies development of macros that are typically heavy on AST manipulations.

2.1 Macro defs

Macro defs are used to morph programs during the compile-time. Whenever a compiler sees an invocation of a method declared as a macro def, it calls the implementation of the macro with the arguments being the ASTs that correspond to the arguments of the original invocation. After the macro returns, its result gets inlined into the call site.

The following example describes a type-safe `printf` function. Being a macro, `printf` formats the output during the compile-time, which provides safety guarantees inaccessible to regular approaches. This notion can be generalized to serve the needs of many other domains, the topic that is dwelled upon later in this section when discussing quasiquotations.

```
macro def printf(format: String, params: Any*) {
  val (evals, refs) = parse(format, params)
  val seq = evals + refs.map(x => scala"print($x)")
  scala"$seq"
}

printf("Value = %d", 123 + 877)
```

Macro defs are called as regular functions, as illustrated by the example. They run during the compile-time and operates on abstract syntax trees. `format`, `params` and the return value are of type `Tree`, which represents Scala ASTs. Macros can call arbitrary functions - for example, `parse` is a normal Scala method.

2.2 Macro types

Macro types are used to parametrically generate classes and traits that can be utilized directly, extended or mixed in, similarly to vanilla Scala classes and

traits. Moreover, macro types can also declare new package objects full of definitions to be imported using regular Scala import syntax.

After creating a suitable macro type for such domains as database access, inter-process interoperability, web services, the programmer is relieved from the necessity to generate and foster boilerplate classes. Since macro types integrate into Scala type system, they provide more flexibility than ad-hoc approaches to these problems.

```
macro trait MySqlDb(connString: String) = ...
type MyDb = Base with MySqlDb("Server=127.0.0.1;Database=Foo;")

import MyDb._
val products = new MyDb().products
products.filter(p => p.name.startsWith("foo")).toList
```

Macro types are, in essence, very similar to macro defs - they are also functions that take ASTs and produce ASTs. Macro types generate fields, methods, inner classes and whatever members are supported by Scala. As shown in the code snippet, this virtual code can be used as if it were written manually.

2.3 Macro annotations

Macro annotations can be used to perform postprocessing on program elements (classes, methods, expressions, etc). Macro annotations are written in the very same way as regular annotations do, however, they are not just static participants of the compilation pipeline, but can actively participate in it.

Potential areas of applicability of this feature include aspect-oriented programming, implementing new idioms (e.g. with macro annotations it's possible to introduce `@lazy` parameters or even work on sub-method level to, say, bless certain variables) and automatizing chores on a small scale.

The example below features the `@Serializable` macro that reflects upon the members of the annotated class, generates serialization/deserialization routines and implements the `Serializable` interface that exposes them (compiler API used here is completely fictional, and is provided solely for demonstrational purposes).

```
macro annotation Serializable(implicit ctx: AnnotationContext) = ...

@Serializable
case class Person(name: String) extends Entity { ... }
```

2.4 Quasiquotations

Macros introduced in previous sections are all about manipulating abstract syntax trees: most of their parameters are ASTs, they manipulate ASTs and, finally,

their return values are ASTs. However, even in languages with algebraic data types and pattern-matching, dealing with ASTs is cumbersome and tedious.

This situation can be remedied by quasiquotations, a DSL for AST manipulation. Quasiquotes let the programmer express abstract syntax trees in the very language these trees describe and also provide support for composition and pattern-matching of ASTs.

The first and foremost function of quasiquoting is to transform literal strings into full-fledged abstract syntax trees (this and other code snippets in this subsection print out ASTs in internal format of `scalac` described in [19]).

```
val two = scala"2"
Literal(Constant(2))
```

Quasiquotations also provide means to embed smaller ASTs into larger ones by the means of splicing. When mentioned in a splice, a variable or an expression that evaluates to an AST will be inserted into the resulting quasiquote.

```
val four = scala"$two + $two"
Apply(Select(two, newTermName("$plus")), List(two))
```

Finally, quasiquotes can be used in pattern matching against abstract syntax trees. In that case splices in quasiquote patterns denote variables that will be bound during the match.

```
four match { case scala"2 + $x" => println(showRaw(x)) }
Literal(Constant(2))
```

Finally, the mechanism of quasiquotations as processed strings can be generalized to embrace arbitrary external domain-specific languages. By implementing parsing, splicing and possibly pattern-matching for a DSL, the programmer enjoys tight integration of the DSL with Scala. String processors, being implemented as macros, can use compiler services (symbol tables, typechecker, etc) and perform precompilation, which proves useful in real-world programming tasks [18].

3 Progress and future work

Project Kepler has been developed as a fork of the Scala compiler [20]. At the moment, prototypes of macro defs and generalized quasiquotations are implemented in the project's public repository [21].

Since its inception the project has received positive feedback from the community [22]. It is a great pleasure for me to mention that Martin Odersky, the creator of Scala, has used macros to improve performance characteristics of `Range.foreach` from Scala standard library [23].

Our future work involves implementing the rest of the proposal and formalizing it as an extension to Scala. Other research areas include employing compile-time metaprogramming facilities to implement type manifests [24] and to enable structured means of type-level programming.

References

1. Czarnecki, K., Donnell, J. O., and Taha, W., *DSL Implementation in MetaOCaml, Template Haskell and C++*. DomainSpecific Program Generation, 2004.
2. Seefried, S., Chakravarty, M., and Keller, G., *Optimising Embedded DSLs using Template Haskell*. Generative Programming and Component Engineering, 2004.
3. Odersky, M., Spoon L., and Venners B., *Programming in Scala, Second Edition*. Artima Press, 2010.
4. Odersky, M., and Zenger M., *Scalable Component Abstractions*. ACM Sigplan Notices, 2005.
5. Odersky, M., and Moors, A., *Fighting Bit Rot with Types (Experience Report: Scala Collections)*. Theoretical Computer Science, 2009.
6. Box, D., and Hejlsberg, A., *LINQ: .NET Language-Integrated Query*, Retrieved from http://msdn.microsoft.com/en-us/library/bb308959.aspx, 2007.
7. Hale C., *Private correspondence with Donald Fischer and Martin Odersky*, Retrieved from http://codahale.com/downloads/email-to-donald.txt, 2011.
8. Dragos I., *Optimizing Higher-Order Functions in Scala*, Third International Workshop on Implementation Compilation Optimization of ObjectOriented Languages Programs and Systems, 2008.
9. McCool, M. D., Qin, Z., and Popa, T. S., *Shader metaprogramming*, Proceedings of the ACM SIGGRAPHEUROGRAPHICS conference on Graphics hardware, 2002.
10. Sheard, T., and Peyton Jones, S., *Template Meta-programming for Haskell*, Haskell Workshop, 2002.
11. Skalski K., *Syntax-extending and type-reflecting macros in an object-oriented language*, Master Thesis, 2005.
12. Scala Macros, *Use cases*, Retrieved from http://scalamacros.org/usecases.html, 2012.
13. Attardi, G., and Cisternino, A., *Reflection support by means of template metaprogramming*, Time, 2001.
14. 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.
15. Steele, G., *Common LISP. The Language. Second Edition*, Digital Press, 1990.
16. Spoon, L., *Writing Scala Compiler Plugins*, Retrieved from http://www.scala-lang.org/node/140, 2008.
17. Bawden, A., *Quasiquotation in Lisp*, Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and SemanticsBased Program Manipulation, 1999.
18. Mainland, G., *Why it's Nice to be Quoted: Quasiquoting for Haskell*, Applied Sciences, 2007.
19. Stocker, M., *Scala Refactoring*, Master Thesis, 2010.
20. The Scala Programming Language, *The Scala programming language repository, GitHub*, Retrieved from https://github.com/scala/scala, 2012.
21. Scala Macros, *Project Kepler: Compiler-Time Metaprogramming for Scala, GitHub*, Retrieved from https://github.com/scalamacros/kepler, 2012.
22. Scala-user mailing list, *Implementing macros for Scala, looking for feedback!*, Retrieved from http://groups.google.com/group/scala-user/browse_thread/thread/800353f4a9ce36b9, 2011.
23. The Scala Programming Language, *Replace Range.foreach by a while loop.*, Retrieved from https://github.com/odersky/scala/commit/35b36229b189a, 2012.
24. Dubochet, G., The Scala Programming Language, *Embedded Domain-Specific Languages using Libraries and Dynamic Metaprogramming*, PhD Thesis, 2011.