MorphScala: Safe Class Morphing with Macros

Aggelos Biboudis
University of Athens
biboudis@di.uoa.gr

Eugene Burmako
EPFL, Switzerland
eugene.burmako@epfl.ch

ABSTRACT

The goal of this paper is to design an easy type-safe metaprogramming API for Scala to capture generative metaprogramming tasks that depend on existing definitions to generate others, by writing meta-code as close as possible to regular Scala code.

`MorphScala`, is a simple domain specific language based on the for-comprehension syntax, that introduces the *class morphing* metaprogramming technique to the Scala programming language. Class morphing is a flavor of compile time reflection (CTR) over fields or methods of classes and interfaces.

The enabling technologies for `MorphScala` are Scala macros and quasiquotes that provide a powerful API for compile-time transformations.

Categories and Subject Descriptors

D.3.4 [**Programming languages**]: Processors—*Code generation*; D.3.2 [**Programming Languages**]: Language Classifications—*Extensible languages*

General Terms

Languages

Keywords

scala, class morphing, macros, quasiquotes, metaprogramming

1. INTRODUCTION

Elements of modern metaprogramming gained inspiration from metaobject protocols [9] (MOPs) like the metaobject protocol of `Smalltalk` and the Common List Object System (CLOS).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Scala '14, July 28-29, 2014, Uppsala, Sweden.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2868-5/14/07$15.00.
http://dx.doi.org/10.1145/2637647.2637650 .

Metaobjects historically are the basis of many introspection mechanisms that modern programming languages include, either at compile time or at runtime like simple type introspection, reflection, macros, quasiquotes or of higher level ones like `F#`'s type providers [16]. The technique named *Computational Reflection* in `Smalltalk-80` enabled, through MOPs, the restructuring of classes, the creation and deletion of methods, the evolution of new class hierarchies at runtime or even adding the support of delegation at runtime [5].

Metaclasses are classes whose instances are other classes. A metaobject protocol is a set of rules that dictates the creation of classes. Several mainstream programming languages include the notion of metaclasses today such as Common Lisp, Groovy, Python, Ruby. In this work, we introduce the *class morphing* technique as our metaobject protocol to the Scala programming language, gaining inspiration from `MorphJ` [6] for Java. `MorphJ` enables static inspection and generation of classes in a type-safe fashion.

`MorphScala` introduces type-safe aspect oriented programming [11, 10] (AOP) to Scala. In AOP one can program aspect advices for structural alterations, such as before-, after-, and around-advices. AOP frameworks are based on source- or bytecode-weaving. Advice languages typically do not take advantage of static type guarantees. This approach, while straightforward, leads to an explosion of developers' assumptions during development [17].

`MorphScala`'s API is defined in terms of the well- established Scala Collection API. Programmers' intuition when using methods for traversal like `foreach`, `filter`, `forall` is reused in `MorphScala` for the purpose of traversing AST elements, filtering them via a pattern-based API and providing definitions of methods and fields in matching cases. The benefits of this approach are that a) programmers can use such an API in the same fashion as they would use collections, b) our API enables the user to organize cross-cutting behaviors in standalone metaclasses, in a type-safe manner c) implementation-wise `MorphScala` relies on macros [2] (since Scala 2.10) and quasiquotes [14] that solve the problem of manual tree construction and deconstruction. Macros make the implementation of `MorphScala` independent of internal compiler API's and quasiquotes are used as part of both the language to develop metaclasses and of the internal implementation of the library.

In this paper:

- We present a sketch of the design of metaclasses in `MorphScala`.
- We present use cases that demonstrate what static safety means at the metaclass level.

- We discuss the implementation strategy of MorphScala.

Our work is currently in the design stage, with full implementation to follow. In this paper, we demonstrate how MorphScala will be used from the user's perspective and we initiate a discussion on our goal for type-safe quarantees at the metaclass level.

2. OVERVIEW AND USAGE

Metaclasses in MorphScala resemble regular class definitions. However, metaclasses are never instantiated with `new` - their role is to describe templates for code generation. Metaclasses in MorphScala are designated by the `@morph` annotation that signifies that the annottee is going to describe one or more *rules* for generating other classes. A rule is expressed through a *reflective block declarations*, and it comprises a pattern definition that will filter members from input type definitions (either coming from generic type parameters or concrete types) and create new definitions based on the input ones. A reflective block is declared as a `for`-comprehension with the range of the iteration designated with `members[X]`, where `X` is a concrete type or a type parameter of the metaclass, the underlying patterns being quasiquotes and the body of the block being the template for newly generated members. The body of the reflective definition block is also specified as a quasiquote, which allows for quite a concise notation.

In the following example the reflective block definition is in line 3 and its body is declared as a quasiquote in lines 4-6. The pattern contains a metavariable `m` indicated with the dollar sign. With quasiquotes metavariables can capture names, terms, types, parameters and even lists of those.

```
1  @morph
2  class Metaclass[X] {
3    for(q"def $m(): Unit" <- members[X]) {
4      q"""def $m(): Unit = {
5            println("Hello World");
6          }"""
7    }
8  }
```

This class describes a different structure for each instantiation. For every concrete type that will substitute `X`, the set of its methods that return `Unit` will be discovered and for each one, a new one will be generated with the same signature but with a single `println` call. Essentially, for every different instantiation, such as `Metaclass[Integer]` or `Metaclass[Customer]`, the corresponding instance will be equipped with a variable set of operations that depend on the generic argument.

```
1  @morph
2  class Log[X] extends X {
3    for(q"def $m(..$params): $r" <- members[X]) {
4      val args = params.map(p => q"${p.name}")
5      q"""override def $m(..$params): $r = {
6            val result = super.$m(..$args)
7            println(result)
8            result
9          }"""
10   }
11 }
```

In the example above, the metaclass `Log` describes a mixin metaclass the goal of which is to enhance instances with logging capabilities. In the `AspectJ` world this is equivalent to performing a *before* action at a *join point* matched by a *pointcut*.

On line 2, a type parameter appears as a supertype of the class, indicating that e.g., a `Customer` class that is going to be declared as `LogMe[Customer]` is going to be a subtype of the `Customer` class. This style of mixin programming, often referred to as: an *abstract subclass* parameterized by its superclass [1], is something that Scala mixins don't support yet (in fact, if not for the `@morph` annotation, this definition would be invalid in Scala, because inheritance from type parameters is prohibited).

On line 3, we introduce the reflective block declaration, which captures all methods that have non-zero parameters and a return type. The rule generates a method with the same signature for each captured method. Each will invoke the corresponding method from the parent class, thus delegating the call, print the result and return it. Late binding is supported as the generated methods come from static reflection over `X`'s methods which are also visible to the original (unmorphed) class that is a supertype of `Log`, as expected.

```
1  @morph
2  class Pair[X, Y](x: X, y: Y) {
3    for(q"def $m1(..$params1): $r1" <- members[X];
4        q"def $m2(..$params2): $r2" <- members[Y];
5        if m1 == m2 &&
6           params1 == params2 &&
7           r1 == r2) {
8      val args = params1.map(p => q"${p.name}")
9      q"""def $m1(..$params1): Pair[X, Y] =
10           new Pair(x.$m1(..$args),
11                    y.$m2(..$args))"""
12   }
13 }
```

In the last example of this section we present a metaclass that contains two patterns; correlating the pattern matching over `X` with `Y`. This means that methods that are matched from the first pattern must also exist in the second. The `Pair` metaclass will contain all common methods with the same signature.

3. TYPE CHECKING METACLASSES

In this section we describe informally the basic rules that need to be applied statically that are going to ensure that metaclasses are valid. Our goal is to explore how our design will enable type checking, without relying on expansion of macros at the instantiation point. Type errors are caught after the expansion of macros, essentially when the macro returns. In order to have type checking at the metaclass level we want to introspect metaclasses as early as possible and reason about validity of emitted code (without emitting it first). Validity for one metaclass means that several reflective blocks will generate non-conflicting method definitions. Validity for several metaclasses means that inter-metaclass references are valid by only looking at metaclasses and not their instantiations.

3.1 Uniqueness of Declarations

MorphScala is able to detect statically if two metaclasses are well-formed. Well-formedness in metaclasses means that `for` declarations do not produce conflicting definitions. In the following example, `CopyMethods` has one type parameter and the single reflective declaration generates method definitions as captured by the pattern. This metaclass is well-formed because for every well-typed `X` it will always produce a well-typed result.

```
1  @morph
2  class CopyMethods[X] {
3    for(q"def $m(..$params): $r" <- members[X]) {
4      q"""def $m(..$params): $r = {
5            ...
6          }""""
7    }
8  }
```

In another example below, the two reflective declarations on lines 3 and 9 can be shown to not overlap, because the number of parameters in methods that they generate is different.

```
1  @morph
2  class DisjointMethods[X] {
3    for(q"def $m($a: Int): $r" <- members[X]) {
4      q"""def $m($a: Int): $r = {
5            ...
6          }"""
7    }
8
9    for(q"def $m($a: Int): $r" <- members[X]) {
10     q"""def $m($a: Int, s: String): $r = {
11           ...
12         }"""
13   }
14 }
```

3.2 Validity of Cross References Between Metaclasses

In the following example we present two metaclasses. Each contains only one definition which is a reflective block definition. The second also contains a reference to the first. An additional challenge in validating well-formedness of these metaclasses is proving that references to methods matched by the patterns are guaranteed to be well-typed. In our case, `UnitPair` is parameterized by two types `X` and `Y` and captures the following pattern: this class contains all methods of `X` that take any number of arguments and return `Unit` that also exist with the same signature in `Y`. Each method that is reflected by this pattern will contain a block that calls the same method in two different objects of the same types that were iterated over.

```
1  @morph
2  class UnitPair[X, Y](x: X, y: Y) {
3    for(q"def $m1(..$params1): Unit" <- members[X];
4        q"def $m2(..$params2): Unit" <- members[Y];
5        if m1 == m2 && params1 == params2) {
6      val args = params1.map(p => q"${p.name}")
7      q"""def $m1(..$params1) = {
8            x.$m1(..$args)
9            y.$m2(..$args)
10         }"""
11   }
12 }
```

`CallUnitWithString` defines a metaclass that has a reference to `UnitPair`. Similarly it defines a pattern but for methods that take a single `String` parameter.

```
1  @morph
2  class CallUnitWithString[T, S](t: T, s: S) {
3    val UnitPair: UnitPair[T, S] = ...
4    for(q"def $n1($s: String): Unit" <- members[T];
5        q"def $n2($s: String): Unit" <- members[S];
6        if n1 == n2) {
7      q"""def $n1($s: String): Unit = {
8            UnitPair.$n1($s)
9          }"""
10   }
11 }
```

Additionally, `CallUnitWithString` and `UnitPair` share the same type parameters. We also notice that the patterns in lines 3-4 are less specific than the pattern in 17-18. This means that by unification, methods captured from the most specific pattern will definitely exist in `UnitPair` as well. Therefore, it is statically safe to call methods in the `UnitPair` instance in this manner.

4. USES CASES

In the following paragraphs we briefly describe potential usages of `MorphScala`.

Proxies of libraries.

Usually a library must be accessed under certain assumptions. For example, a collection library must be accessed under the assumption of synchronized access. A family of metaclasses that capture the essence of a design, by being immune to interface/signature changes (due to pattern-based reflection), can prove beneficial and error-free by automating the generation of proxies.

Testing/benchmarking frameworks.

When using an automated testing framework it is often needed to employ the same testing activity for many operations. As methods and fields of a class definition evolve (added/removed), the corresponding testing/benchmarking code must be maintained separately. `MorphScala` may be used side-by-side with such Scala frameworks.

Mocking frameworks.

Mocking frameworks usually employ reflection for dynamic generation of mock objects or are implemented as compiler plugins to enable compile-time generation like ScalaMock 2 [13]. ScalaMock 3 uses macros for compile-time proxies. The API behind mock objects generation can be captured with `MorphScala`.

5. IMPLEMENTATION DISCUSSION

The implementation scheme we describe in this section is constituted by two phases of macro expansion. Our aim is to keep the metaclass as simple as possible. The metaclass is annotated with a *macro annotation* [3] with the syntax described above. The first expansion (of a macro annotation) transforms this class into a type definition that uses *type macro* [4]. Additionally, in this phase, type checking takes place. The `@morph` annotation can save the entire AST of the annottee and store it for later use (e.g. as a non-macro annotation on the resulting type macro). All the AST's of annottees, constitute the metaclasses that need to be type-checked for validity of references and validity of declarations.

The second expansion (of a type macro) that takes place much later, generates code from the template of the metaclass expressed as a type macro and its input type arguments.

In `MorphScala` we use the familiar `for` syntax ranging over member definitions statically. The type definition that we want to range over, ideally is a collection of members that satisfy the pattern. For this reason we use a generic method, which we call `members`, that represents the collection of such definitions at compile time. In order to use `members` as a part of our for-comprehension syntax, its return type must support methods like `foreach`, `filter`, etc.

The `members` method is just a stub. Its sole purpose is to provide a placeholder, which our macros will detect and use to drive the translation. If the user only calls `members` and does not call any macros on it (which does not make much sense in CTR context), that's going to be a static error. Consider our toy example again, `Log[X]`, instantiated as `Log[Customer]` with a `Customer` object that has one method named `order`. The simplified translation will be the following:

```
members[Customer]
    .foreach((quasiquote) => (quasiquote));
```

`foreach` will be supported as a macro definition on the return type of `members` and its purpose will be to expand to method declarations using the passed quasiquote as template. This is the second phase of macro expansion.

The interesting part is that after expansion, a new synthetic type definition must be generated. Type macros rewrite a `new Log[Customer]` use-site to an expression using the synthetic/specilized class `new Log$Customer` whose concrete type definition will effectively be `Log$Customer extends Customer`.

6. RELATED WORK

With Scala mixins one can achieve an interceptor-like style of programming by composing mixins together, emulating effectively the cross-cutting concerns from AOP [8] using the Dynamic Proxy API of Java.

Method Proxy-Based AOP in Scala [15] uses similar syntax for pointcuts and advices as AspectJ. Method call interception is based on an `AOP` trait, that acts as a central database of aspects, and an `Aspect` class. This work is based on delegation of higher-order methods, managed by AOP proxies. It features some form of static-checking, relying on Scala's type system, however no inter-aspect guarantees are discussed. Additionally, the dynamic-dispatch nature of proxies imposes an execution overhead on any adviced code. Both solutions appeared before Scala macros.

7. FUTURE WORK

In this paper we present the design of `MorphScala`. Two areas that will need investigation are the type-checking rules and type macros. Additionally we propose two interesting directions about patterns and generation.

Type checking.

We can regard `MorphJ` as a starting point for the static typing of `MorphScala`. We refer the reader to [6, 7] for a more in-depth discussion of `MorphJ`'s typing rules. However, Java has use-site variance annotations through wildcards, while Scala has also declaration-site variance (wildcards are still supported (e.g. `Log[_]`). This imposes a question on how this will affect the typing rules for `MorphScala`.

Type Macros.

Type macros were introduced in Scala as an experimental addition to macros [4]. Unlike def macros, which expand into terms, type macros expand into types, which makes them ideal for our use case. Types generated by type macros can be introduced as a top level definition that is maintained by the compiler. This was the most controversial bit of the design of type macros, because it led to dangerous non-localities in macro expansions (causing non-determinism in builds and hard-to-understand dependencies between program elements), which ultimately led to their deprecation and removal from Macro Paradise [1], the experimental build of Scala compiler that provides additional macro facilities. However, we believe that this functionality can be reintroduced into Macro Paradise in a disciplined manner.

Compile time reflection over additional constructs.

At the moment, `MorphScala` only supports iteration over methods and fields. Iterating over other sorts of definitions (e.g. inner classes and objects) would increase expressivity of our library in contrast to MorphJ.

Generating Expressions.

In [12] the authors take an interesting direction of compile-time reflection. Apart from method/field definitions, they present a pattern-based approach generating statements as well. We believe that this approach will generalize the concept of pattern-based reflection up to the expression level in the context of `MorphScala`.

8. SUMMARY

In this paper we present a design strategy for `MorphScala`, an informal discussion of the cases that a set of type checking rules must capture for type safety and use cases that can benefit from our library. We attempt to keep the API as close as possible to MoprhJ in valid Scala syntax powered by macros and quasiquotes.

Acknowledgments

We would like to thank Denys Shabalin, Gian Ntzik and Nick Palladinos for their valuable comments that helped strengthen the paper. We gratefully acknowledge funding by the Greek Secretariat for Research and Technology under an Excellence (Aristeia) award co-funded by the European Social Fund (ESF) and National Resources (MORPH-PL.)

9. REFERENCES

[1] G. Bracha and W. Cook. Mixin-based Inheritance. In *Proc. of the European Conference on Object-oriented Programming on Object-oriented Programming Systems, Languages, and Applications (OOPSLA/ECOOP)*, volume 25, pages 303–311, Ottawa, Canada, 1990.

[2] E. Burmako. Scala Macros: Let Our Powers Combine!: On How Rich Syntax and Static Types Work with Metaprogramming. In *Proc. of the 4th Workshop on Scala*, page 3, Montpellier, France, 2013. ACM.

[3] Eugene Burmako. Macro Annotations. `http://docs.scala-lang.org/overviews/macros/annotations.html`.

[1]http://docs.scala-lang.org/overviews/macros/paradise.html

[4] Eugene Burmako. Type Macros. http://docs.scala-lang.org/overviews/macros/typemacros.html.

[5] B. Foote and R. E. Johnson. Reflective Facilities in Smalltalk-80. In *Proc. Conf. on Object-Oriented Programming Systems, Languages and Applications*, OOPSLA '89, pages 327–335, New York, NY, USA, 1989. ACM.

[6] S. S. Huang and Y. Smaragdakis. Expressive and safe static reflection with MorphJ. In *Proc. Conf. on Programming Language Design and Implementation (PLDI)*, volume 43, pages 79–89, Tucson, AZ, USA, 2008. ACM.

[7] S. S. Huang and Y. Smaragdakis. Morphing: Structurally Shaping a Class by Reflecting on Others. *ACM Transactions on Programming Languages and Systems*, 33(2):1–44, Feb. 2011.

[8] Jonas Boner. Real-World Scala: Managing Cross-Cutting Concerns using Mixin Composition and AOP. http://jonasboner.com/2008/12/09/real-world-scala-managing-cross-cutting-concerns-using-mixin-composition-and-aop/, Dec. 2008.

[9] G. Kiczales. *The Art of the Metaobject Protocol*. MIT press, 1991.

[10] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proc. European Conference on Object-Oriented Programming (ECOOP)*, pages 327–353, London, UK, 2001.

[11] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proc. European Conference on Object-Oriented Programming (ECOOP)*, volume 1241, pages 220–242. Springer, Heidelberg, Germany, and New York, 1997.

[12] W. Miao and J. Siek. Compile-time Reflection and Metaprogramming for Java. In *Proc. of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation*, pages 27–38, San Diego, California, USA, 2014. ACM.

[13] Paul Butcher. ScalaMock: Native Scala mocking. http://scalamock.org/.

[14] D. Shabalin, E. Burmako, and M. Odersky. Quasiquotes for Scala. Technical report, 2013.

[15] D. Spiewak and T. Zhao. Method Proxy-Based AOP in Scala. *Journal of Object Technology*, 8(7):149–169, 2009.

[16] D. Syme, K. Battocchi, K. Takeda, D. Malayeri, J. Fisher, J. Hu, T. Liu, B. McNamara, D. Quirk, and M. Taveggia. Strongly-Typed Language Support for Internet-Scale Information Sources. Technical report, Technical Report MSR-TR-2012-101, Microsoft Research, 2012.

[17] S. Zschaler and A. Rashid. Aspect Assumptions: A Retrospective Study of AspectJ Developers' Assumptions About Aspect Usage. In *Proc. of the Tenth International Conference on Aspect-oriented Software Development*, AOSD '11, pages 93–104, New York, NY, USA, 2011. ACM.