## MorphScala: Safe Class Morphing with Macros

Aggelos Biboudis (@biboudis) Eugene Burmako (@xeno\_by)

University of Athens École Polytechnique Fédérale de Lausanne

29 July 2014

### Outline

- Motivation for class morphing in Scala
- ▶ The idea presented in the paper (the implementation is not there yet!)
- ► Future implementation steps

## Motivation

### Scala macros

- ► Experimental language feature available for almost 2 years
- Multiple flavors: def macros, type macros, macro annotations
- Most widely used are def macros: methods whose applications expand on sub-method level

### Blackbox macros

```
trait Query[T] {
  def filter(p: T => Boolean): Query[T] = macro ...
}

val users: Query[User] = ...
users.filter(_.name == "John")
```



Query(Filter(users, Equals(Ref("name"), Literal("John"))))

- Look like normal methods with honest type signatures
- Can be treated as black boxes by machines and humans alike

### Whitebox macros

```
def h2db(connString: String): Any = macro ...
val db = h2db("jdbc:h2:coffees.h2.db")
val db = {
  trait Db {
    case class Coffee(...)
    val Coffees: Table[Coffee] = ...
 new Db {}
```

- Can refine their return types
- ▶ Thanks to first-class modules this can generate public definitions

### The whitebox conundrum

- ▶ Practice shows that type signatures are very important for macros
- ▶ But practice also demands intra-method code generation
- Blackbox can't do the latter, whitebox isn't very good at the former
- Stalemate?

## A new hope

- ► At last year's OOPSLA, Aggelos introduced me to MorphJ
- MorphJ is an extension to Java that enables class morphing, a form of intra-method template metaprogramming (TMP)
- ► TMP is an interesting declarative approach to metaprogramming, but most importantly MorphJ's TMP allows modular typechecking
- So we decided to give it a try

The idea

### Metaclasses

#### @morph

```
class Logged[X] extends X {
  for (q"def $m(..$params): $r" <- members[X]) {
    val args = params.map(p => q"${p.name}")
    q"""override def $m(..$params): $r = {
        val result = super.$m(..$args)
        println(result)
        result
    }"""
  }
}
```

- @morph designates a metaclass for Logged[X] classes
- ▶ Logged[X] classes are instantiated and used in the normal fashion

## Compile-time reflection

```
@morph
class Logged[X] extends X {
  for (q"def $m(..$params): $r" <- members[X]) {
    val args = params.map(p => q"${p.name}")
    q"""override def $m(..$params): $r = {
          val result = super.$m(..$args)
          println(result)
          result
        }"""
```

- MorphScala provides a quasiquote-based DSL to iterate members
- ▶ Members can be destructured and give rise to new members

## (1) Uniqueness of declarations

```
@morph
class Logged[X] extends X {
  for (q"def $m(...$params): $r" <- members[X]) {
    val args = params.map(p => q"${p.name}")
    q"""override def $m(..$params): $r = {
          val result = super.$m(..$args)
          println(result)
          result
        }"""
```

- A very appealing property of morphing is modular type checking
- ▶ First, we want to guarantee that generated members don't overlap

# (2) Validity of references

```
@morph
class Logged[X] extends X {
  for (q"def $m(...$params): $r" <- members[X]) {
    val args = params.map(p => q"${p.name}")
    q"""override def $m(..$params): $r = {
          val result = super.$m(..$args)
          println(result)
          result
        }"""
```

- A very appealing property of morphing is modular type checking
- Second, we want to guarantee that all references are valid

## Conclusion

### Status

- ▶ We haven't implemented the outlined idea yet
- ▶ There are still some open questions
- ▶ However most of the implementation tools are already in place

# (1) Restrictions for the compile-time reflection API

```
@morph
class Logged[X] extends X {
  for (q''def \m(...\params): \m' <- members[X]) {
    val args = params.map(p => q"${p.name}")
    q"""override def $m(..$params): $r = {
          val result = super.$m(..$args)
          println(result)
          result
        }"""
```

- Modular type checking is very important
- How much flexibility do we have to sacrifice to achieve it?
- What constructs can we safely intrinsify?

# (2) Implementation vehicles

```
@morph
```

```
class Logged[X] extends X {
  for (q"def $m(..$params): $r" <- members[X]) {
    val args = params.map(p => q"${p.name}")
    q"""override def $m(..$params): $r = {
        val result = super.$m(..$args)
        println(result)
        result
    }"""
  }
}
```

- ▶ Can be implemented by a combo of type macros and macro annots
- Are we happy about this elaborate scheme?
- How do we revive type macros?

## Summary

- MorphScala = template metaprogramming facility inspired by MorphJ
- Emphasis on strong type checking guarantees (modular type checking)
- ▶ Implementation is in the works