Scala Macros

Eugene Burmako Martin Odersky

École Polytechnique Fédérale de Lausanne

09 July 2012 / Meta 2012

What is this talk about?

- Compile-time metaprogramming
- Type-safe AST transformers (called "macros" in Scala and in several other languages)
- Road to macros in Scala

Behind the scenes

- Advanced features of Scala's type system w.r.t macros
  - Cross-stage path-dependent types
  - Type inference in presence of macros
  - Implicits in macro declarations and implementations
- Design of the Scala reflection library
  - Cake pattern to provide different views into the compiler
  - Abstract types that enable virtual classes
  - Uniformity of compile-time and runtime reflection

Outline

Project Kepler

The project was started in October 2011 with the following goals in mind:

- To democratize metaprogramming (at the moment there's a lot of hype that the future is multicore; along the similar lines my belief is that the future is meta).
- To solve several hot problems in Scala: insufficient control over inlining, need for reification in domain-specific languages.

Since April 2012 (milestone pre-release 2.10.0-M3) macros are a part of Scala. Several practical (data access facility, unit testing framework, library for numeric computations) and research projects are already using macros.

Macros in Scala

```
def assert(cond: Boolean, msg: Any) = macro impl
def impl(c: Context)(cond: c.Expr[Boolean], msg: c.Expr[Any]) =
  if (assertionsEnabled)
    // if (!cond) raise(msg)
    If(Select(cond.tree, newTermName("$unary_bang")),
        Apply(Ident(newTermName("raise")), List(msg.tree)),
        Literal(Constant(())))
  else
    Literal(Constant(())

import assert
assert(2 + 2 == 4, "weird arithmetic")
```

- Metalanguage = target language
- Macros work with ASTs rather than with text
- Type awareness and type-safety

Putting macros in perspective

- Text generators (C/C++ preprocessor, M4). No integration into grammar or semantics of the target language.
- Syntax extenders (CamlP4, SugarHaskell, Marco). Define new productions and non-terminals for the original grammar. Rely on ad-hoc tricks to get semantic information (bindings, types, etc).
- Deeply embedded DSLs (Virtualized Scala, LMS). Reuse host parser and typechecker, yet can override semantics.
- Macros (LISP, Scheme, MacroML, Template Haskell, *Nemerle*, etc). Integrated into the compiler, expand during compilation, typically have access to the compiler API.
- Metalanguages (N2). Every aspect of a language (parser, type checker, code generation, IDE integration) is customizable.

Why this talk is interesting

- Metacomputations (because metaprogramming is cool)
- Linguistics (several notational problems and solutions w.r.t metaprogramming)
- Metamacros (macro-generating macros, notation macros, self-cleaning macros)

Outline

Notation for macro triggers

Function application (very traditional, implemented):

```
macro def assert(cond: Boolean, msg: Any) = ...
assert(2 + 2 == 4, "weird arithmetic")
```

Type construction (a natural desire for a typed language, planned):

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

Declaration of program elements (tentative):

```
macro annotation Serializable = ...
@Serializable class Person(...)
```

Choice of macro triggers is arbitrary and ad-hoc. To do better we need integration with the parser (Nemerle, SugarHaskell).

Notation for metacode

The first approach (similar to Template Haskell and Nemerle):

```
macro def assert(cond: Boolean, msg: Any) =
  if (assertionsEnabled)
    If(Select(cond, newTermName("$unary_bang")),
        Apply(Ident(newTermName("raise")), List(msg)),
        Literal(Constant(())))
  else
    Literal(Constant(()))
```

- Minimalistic and appealing at a glance
- Transparent to the user, as the signature doesn't reveal the underlying magic
- Cross-stage lexical scoping is very potent
- Too potent to be robust

Problem with the first approach

```
class Queryable[T, Repr](query: Query) {
  macro def filter(p: T => Boolean): Repr =
    Apply(Ident(newTermName("Query")),
      List(Apply(Ident(newTermName("Filter")),
        List(query, reify(p)))))
}
```

- *p* being used in a macro expansion is okay, since it comes from the same metalevel.
- But what about *query*? This is a runtime value, so we cannot splice it into the result of macro expansion.
- Adapting closure conversion we could make it work, but that would bring significant technical and cognitive problems.
- To avoid this problem Template Haskell and Nemerle only allow macros in top-level definitions!

Notation for metacode

The second approach:

```
def assert(cond: Boolean, msg: Any) = macro impl
def impl(c: Context)(cond: c.Expr[Boolean], msg: c.Expr[Any]) =
  if (assertionsEnabled)
    If(Select(cond.tree, newTermName("$unary_bang")),
        Apply(Ident(newTermName("raise")), List(msg.tree)),
        Literal(Constant(())))
  else
    Literal(Constant(())
```

- Splits macro definitions and macro implementations. The latter are only allowed in static contexts.
- As a pleasant side effect, macro parameter magic is gone, and macro implementations are now first-class.

Notation for quasiquoting

An obvious approach is to introduce new syntax, following multiple languages which have done that:

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

Being obvious this design decision is also suboptimal:

- Adds extra burden on the language spec
- Complicates parsing
- Is opaque to existing tools

Notation for quasiquoting

When macros started brewing, Scala got string interpolation. We generalized the interpolation proposal to accomodate a wide range of syntaxes:

```
scala"if (!$cond) raise($msg)"
```

gets desugared by the parser into the following snippet:

```
StringContext("if (!", ") raise (", ")").scala(cond, msg)
```

- No changes to the compiler
- Modularity and extensibility (anyone can "pimp" the scala method onto StringContext with implicit conversions)
- Partial amenability to automatic analysis

Outline

A discovery

Macro-based string interpolation expressing quasiquotes is nice, being minimalistic, expressive and performant. What's even more important, it's conventional.

```
scala"if (!$cond) raise($msg)"
```

A key insight however was to explore the design space further, which gave us this marvel:

```
reify(if (!cond.eval) raise(msg.eval))
```

Notation for quasiquoting

The final solution:

```
reify(if (!cond.eval) raise(msg.eval))
```

reify is a library macro (but could be implemented by a programmer).

It takes an AST that represents an expression (which in Scala can be even a declaration or a sequence of declarations) and generates a tree that will re-create that AST at runtime.

eval method is a marker that tells *reify* to splice the target expression into the resulting AST.

Hygiene

```
def raise(msg: Any) = throw new AssertionError(msg)

def assert(cond: Boolean, msg: Any) = macro impl
def impl(c: Context)(cond: c.Expr[Boolean], msg: c.Expr[Any]) =
  c.reify(if (!cond.eval) raise(msg))

object Test extends App {
  def raise(msg: Any) = { /* haha, tricked you */ }
  assert(2 + 2 == 3, "no way")
}
```

- *reify* solves the problem of inadvertent name captures
- For example *raise* in the AST produced will bind to the original *raise* no matter where it ends up after macro expansion

Staging

reify can also express staging:

- Brackets are implemented by *reify* itself
- Escape is implemented inside *reify* by treating *eval* functions in a special way
- Run is implicitly carried out by compilation and macro expansion (for nested *reify* calls)

Related work

Taha et al. build a macro system atop a staged language:

- *Macros as Multi-Stage Computations* Ganz, Sabry & Taha, ICFP'01
- *Staged Notational Definitions* Taha & Johann, GPCE'03

We build a staged system atop a macro language.

Reified types

reify saves syntax trees and transfers them to the next metalevel. Exactly the same can be done for types!

With *reify* it becomes possible to inspect type arguments of polymorphic functions and type constructors.

Outline

Summary

- Language = metalanguage
- Macro-enabled Scala = Scala + *macro* keyword + a trigger in typer
- This minimalistic core is enough to express *reify*, which can implement quasiquoting, hygiene and staging
- Reification makes Scala homoiconic
- This is officially a part of Scala