Scala Macros

Eugene Burmako

École Polytechnique Fédérale de Lausanne

09 July 2012 / Meta 2012

What is this talk about?

Compile-time metaprogramming with AST transformers (called "macros" in Scala and in several other languages).

Practice-oriented research we've done in Scala macrology: evolution of our design, problems we've faced and solved, open questions.

Behind the scenes

Applications of advanced features of Scala's type system to macros (cross-stage path-dependent types, type inference in presence of macros).

Design of Scala reflection library, which exposes slices of the compiler API enabling programs to inspect themselves at compile-time and runtime.

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 assertImpl
def assertImpl(c: Context)
    (cond: c.Expr[Boolean], msg: c.Expr[Any]) =
  if (assertionsEnabled)
    <[ if (!$cond) raise($msg) ]>
  else
    <[ () ]>

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

- Metalanguage = target language
- Work with ASTs rather than with text
- Are type-aware and type-safe

Note: quasiquoting syntax used in this snippet is fictional. This is to avoid spoiling the fun from a discovery.

Putting macros in perspective

- Text generators (C/C++ preprocessor, M4). Unaware of semantics of the target language. Typically metalanguage is different from the target language.
- Syntax extenders (CamlP4, SugarHaskell, Marco). Let the programmer define new productions and non-terminals for the original language grammar. Have hard time integrating with the compiler (bindings, type system).
- Deeply embedded DSLs (Virtualized Scala, LMS). Enjoy integration with the typechecker, yet can bend a lot of semantic rules of the host language.
- 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 extension points

Function application:

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

Type construction:

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

Postprocessor for typechecking:

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

Choice of extension points feels arbitrary and ad-hoc. To do better we need integration with the parser (Nemerle, SugarHaskell).

Notation for metacode

v1:

```
macro def assert(cond: Boolean, msg: Any) =
  if (assertionsEnabled)
    <[ if (!$cond) raise($msg) ]>
  else
    <[ () ]>
```

- Minimalistic and appealing at a glance
- Transparent to the user, as the signature doesn't reveal the underlying magic
- Lexical scopes that work across stages look good
- Too good to be true

Problem with notation v1

```
class Queryable[T, Repr](query: Query) {
  macro def filter(p: T => Boolean): Repr = <[
    val b = $newBuilder
    b.query = Filter($query, ${reify(p)})
    b.result
  ]>
}
```

- p being used in the quasiquote is okay, since it comes from the same metalevel.
- But what about query? This is a runtime value, so we cannot splice it into a macro expansion.
- Adapting closure conversion we could make it work, but that would bring significant technical and cognitive problems.
- Neither Template Haskell, nor Nemerle allow that.

Notation for metacode

v2:

```
def assert(cond: Boolean, msg: Any) = macro assertImpl
def assertImpl
    (c: Context)
    (cond: c.Expr[Boolean], msg: c.Expr[Any]) =
  if (assertionsEnabled)
    <[ if (!$cond) raise($msg) ]>
  else
    <[ () ]>
```

- 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.
- Much more verbose and involves elaborate types (can be remedied with macro-generating macros).

Notation for embedded code

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. It adds extra burden on the language spec, complicates parsing, is opaque to existing tools.

Notation for embedded code

When macros started brewing, Scala has experienced an explosion of new ideas after Martin Odersky announced a call for proposals to be included in Scala 2.10.

By hijacking a string interpolation proposal, we have been able to extend it significantly. In a nutshell, code like this:

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

Gets desugared by the parser into the following snippet:

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

Now quasiquoting doesn't require any changes to the compiler, is modular (anyone can "pimp" the scala method onto StringContext with implicit conversions) and is, at least, partially amenable to automatic analysis.

Metamacros

The approach outlined above has a minor deficiency. Parsing of quasiquoted Scala code is done at runtime:

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

But why not make "scala" a macro? Then quasiquotes will be parsed at compile-time, and the overhead will be gone.

This is a petty application of macros, and I'm almost embarassed to use the huge "metamacro" word to describe it, but the pomp helps to feel the significance of the example. It shows that macros trivialize ad-hoc metaprogramming.

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 ditching this good enough solution to explore and find something that's even better:

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

Reified trees

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

Reify is a macro.

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 of the Expr class is marker that tells reify to splice the target expression into the resulting AST.

Reify

Being a convenient way to produce ASTs, reify provides a quasiquoting facility that doesn't require new language features and immediately works with existing tools.

More importantly, reify solves the problem of inadvertent name captures that can happen in macro-generated code (so called hygiene problem). Since reify has full control over the AST it generates, it can perform alpha renaming if necessary.

Metamacros

Apparently macros are sufficiently powerful to bootstrap themselves into non-trivial functionality: quasiquoting and hygiene, which are traditionally introduced as separate features in macro-enabled languages.

Staging

Another curious fact is that reify enables staging.

Macros can implement staging primitives from Walid Taha's MetaML thesis:

- Brackets are implemented by reify
- Escape is implemented inside reify by treating eval functions in a special way
- Run maps onto compilation and macro expansions (for nested reify calls)

Related work

Macros as Multi-Stage Computations Ganz, Sabry & Taha

Staged Notational Definitions Taha & Johann

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

We build a staged system atop a macro language.

Reified types

Reify brings saves syntax trees and brings them to the next metalevel. Exactly the same can be done for types.

There is a good reason to do that except plain curiosity. Type reification can defeat erasure.

Scala (similarly to most functional languages) erases type arguments of polymorphic functions and type constructors. Hence types arguments cannot be inspected at runtime. With macros and reification it becomes possible to partially overcome this limitation.

And again this can help writing macros.

Polymorphic macros

```
def serialize[T](x: T) = macro serializeImpl

def serializeImpl[T: c.TypeTag](c: Context)(x: c.Expr[T])
  // inspects T to find out internal structure of T
  // and generate the serialization code
```

- c.TypeTag is a context bound. Scala translates context bounds into implicit parameters. For example, this one will be compiled down to (implicit evidence$1: c.TypeTag[T]).
- When an implicit parameter to a method is not provided, Scala launches implicit search that traverses scopes from the inside out and ends up at the outermost system scope.
- System scope defines a macro that reifies the requested type, providing runtime representation of T to be used within a macro (or in any other method).

Outline

Summary

- The notion of macros is more interesting than what's implemented in C/C++ preprocessor
- Macros enable and encourage ad-hoc metaprogramming
- Being as ad-hoc as they are, macros are nevertheless powerful enough to implement such fundamental concepts as hygiene and staging

Future work

Rephrasing Philip Wadler. How to make ad-hoc metaprogramming less ad-hoc?

Thanks!

eugene.burmako@epfl.ch

http://scalamacros.org