α-Kepler

Eugene Burmako

EPFL, LAMP

14 January 2012

α-Kepler

Hi folks! My name is Eugene Burmako, since the last fall I've been working in Scala Team and studying at EPFL under the supervision of Martin Odersky.

Today we'll discuss the progress of Project Kepler that hosts the development of **macros** and **quasiquotes** - compile-time metaprogramming tools for Scala.

In my previous talk I elaborated on the theoretical side of macrology, but today we'll mostly cover practical aspects. Therefore before proceeding it might be helpful to skim through the slides of my previous talk.

After a short introduction we'll implement a thingie I've been dreaming about since I've learned about Nemerle - of course, it will be a macro. Our agenda also includes discussing Scala abstract syntax trees w.r.t. macro development. Have fun!

Outline

Somewhat related example

One of the most interesting recent events in Scala world was the discussion of string interpolation.

This functionality is supported by many script languages (bash, Perl, Ruby) and provides a possibility to embed variables from lexical scope directly into string literals.

```
scala> val world = "world"
world: String = world

scala> println(s"hello ${world}!")
hello world!
```

The `s` identifier next to the string is not a typo, it signifies that the string will be interpolated (for the sake of backward compatibility dollars in regular string won't be treated specially, so we need to explicitly distinguish processed strings).

Implementing the interpolator

Despite of its conceptual simplicity, interpolation cannot be implemented as a library, since it operates lexical scope that cannot be manipulated explicitly.

To code up a prototype, we can open `doTypedApply` (function that types method application) and insert a check for the interpolation method. Inside the compiler we've got full access to all the semantic information about the program, and we're going to take advantage of that.

Inside the typer it's possible to acquire the string literal to interpolate and to parse that literal right over there. After that it doesn't take a lot of effort to compose a map of visible variables and to turn the invocation of the `s` method into a vanilla string concatenation.

If you fancy, you can look up the implementation details in my repository, but for now let's go ahead.

Using the interpolator

So far, so good. We've injected custom logic into typer and replaced a call to a marker function with a hand-crafted AST. It sounds scary, but no worries - in a few slides we'll be doing the same in live mode =)

However this implementation strategy leaves an open question of integrating our changes into the compiler. Interpolation looks more or less fundamental to be accepted to the upstream, but project-specific tricks would definitely not make it. Maintaining a custom build of `scalac` isn't super convenient, so we need a principled approach.

Conventional solution to this problem is writing a compiler plugin (for example, CPS transformation for Scala is implemented as a compiler plugin), and here we're getting real close to macros.

The essence of macros

Macros are special kinds of compiler plugins that are automatically loaded from classpath and transform certain program elements (method calls, type references, class/method definitions).

Similarly to the aforementioned interpolator, macros:

- Get executed during the compile-time
- Unlike traditional compiler plugins, get transparently loaded by the compiler, without the need of being wrapped in `Plugin` and `Component`
- Work with expression trees that correspond to the code of the program being compiled
- Have access to internal compiler services (reflection against previously compiled code, type checking and type inference, lexical scopes and so on)

Flavors of macros

Macro defs receive the ASTs of their arguments, are expanded into an AST and get inlined into their callsites:

```
macro def printf(format: String, params: Any*) = ...
printf("Value = %d", 123 + 877)
```

Macro types are types that have their members generated during the compilation:

```
macro class MySqlDb(connString: String) = ...
object MyDb extends MySqlDb("Database=Foo;")
```

Macro annotations perform postprocessing of method and type declarations:

```
macro annotation Serializable(implicit ctx: Context) = ...
@Serializable case class Person(name: String)
```

Feedback

During the few months since the kick-off of Project Kepler we've found quite a few use cases that are significantly simplified by macros.

To name only a few. There is significant interest in using macros to implement queries in O/RMs. I've also heard ideas of using macro annotations and macro types to simplify certain functional programming techniques. A few days ago Martin has implemented a prototype of an optimizer, which uses macros to speed-up `Range.foreach`.

Finally, our lab has obtained a grant from Swiss Commission for Technology and Innovation to implement an improvement to Scala, an improvement that is based on reflection and macros.

Status

To the moment we've implemented a public interface to the compiler and its data structures and realized a prototype that supports macro defs. We're quite happy with the prototype and will continue with following this direction.

We are also actively developing quasiquotes, a domain-specific language for creating and decomposing abstract syntax trees. Most likely, quasiquotes will be implemented as a generalization of string interpolation, though things here are much less clear in comparison with macros.

We have put up scalamacros.org, a site that serves as a centralized source of information regarding our project. In our opinion, of significant interest is its "Use Cases" section that elaborates on the areas of applicability of macros.

Outline

Problem statement

Scala provides several ways of using regexes to extract text fragments of certain shapes.

First of all, one can use vanilla imperative `Match.group`. Secondly, Scala library provides an extractor for regexes, so it's possible to extract matched groups using a pattern match. However, both approaches share the same shortcoming. When using named groups, one has to write the names twice - one time inside a regex (to name a group) and another time when processing a match (to declare a bound variable).

It would be great to have group names automatically mapped onto variables in the lexical scope of a match processor, i.e. to be able to write as follows:

```
val s = "foo=bar"
s.forAllMatches("""^(?<key>.*?)=(?<value>.*)$""",
    println("key = %s, value = %s".format(key, value)))
```

Interactive presentation

You can follow the implementation of a macro in interactive mode. To do that, clone https://github.com/xeno-by/alphakeplerdemo and check out the `skeleton` branch.

```
> git clone git@github.com:xeno-by/alphakeplerdemo.git
> git checkout skeleton
```

Skeleton hosts all the necessary boilerplate, with the macro body left not implemented. Source files have to be compiled by the particular compiler that resides next to the sources, since we will use features that aren't in the upstream yet.

And one more remark. Before version 1.7, Java didn't support named groups inside regexes, so I've coded up a small workaround for that. My approach won't work for in a production setting (it fails on nested groups), but it's good enough for a demo.

Meeting the skeleton

Let's open Rx.scala that defines the macro as an extension method (yeh, it's possible to do that!) with the body featuring three question marks (did you know about that feature?):

```
object Rx {
  ...
  implicit def string2mx(pattern: String): Mx =
    new Mx(pattern)
}

class Mx(val s: String) {
  def macro forAllMatches(pattern: String, f: _): Unit = {
    ???
  }
}
```

Signature of the macro

Macro signature is pretty much vanilla, except one thingie - the underscore that takes the place of a parameter type for `f`.

```
def macro forAllMatches(pattern: String, f: _): Unit
```

The underscore denotes the fact that the parameter won't be typed by the typer and will be passed to the macro verbatim (this notation is very experimental, so there's a good chance that beta release will change it, but so far let's use the underscore).

But why do we need this? Why not write `f: => Unit`, similarly to what is done in regular `foreach` from the collection API?

```
def foreach(f: A => Unit): Unit
```

Macro expansion

Normal routine of expanding a macro is similar to the one that is employed for typing a method call: the arguments get typechecked first, and then the macro gets expanded (what is macro expansion? it is invoking a macro implementation using ASTs that corresponds to its parameters).

Let's study this mechanism for our hypothetical macro:

```
val s = "foo=bar"
s.forAllMatches("""^(?<key>.*?)=(?<value>.*)$""",
    println("key = %s, value = %s".format(key, value)))
```

Processing the pattern is pretty much obvious - it will be typed as a regular string, but what do we do with the function (the expession that involves `println`)? Can we type it?

Typing macro arguments

Actually we cannot type the expession that involves `println`, because it uses `foo` and `bar`, the variables that are not present in lexical scope - they will be magically generated by a macro.

Of course, in our case a human would infer these variables being strings, but to the compiler macros are completely opaque (since they represent dynamic pieces of code). For example, a weird macro could nondeterministically return either strings or ints - what would we do then?

Therefore in this case the compiler has to give up typing the expression that is passed to a macro. Does this mean that macros overrule compile-time type checks?

Type safety of macros

Type safety doesn't get violated, since the AST that is returned by a macro expansion, gets typechecked right away.

Just for fun, here's a snippet from Typers.scala that demonstrates expansion/typechecking logic.

```
if (inExprModeButNot(mode, FUNmode) && tree.symbol !=
    null && tree.symbol.isMacro && !tree.isDef) {
  val tree1 = expandMacro(tree)
  if (tree1.isErroneous) tree1 else typed(tree1, mode, pt)
}
```

Cheating a typechecker feels bad - in our case this might lead to misleading error messages (after being rehashed by a macro, errors in ASTs that represent macro arguments might manifest themselves in wildly different contexts). However, this sacrifice is necessary for flexibility purposes (see a restrospective on Template Haskell for more details).

Test run

Now, when we're settled down with the starting point, let's try executing the code as is and seeing what happens. Undoubtedly, this is going to be something interesting.

```
> scalac -Xmacros Rx.scala Test.scala
Test.scala:5: error: macro implementation not found:
    forAllMatches
```

Oops! The result is though-provoking, but not very inspiring. We've stumbled upon the problem of joint compilation of macros and their clients.

Separate compilation

Here's the deal. As we've mentioned earlier, macros are mini-plugins to the compiler that are loaded from the classpath and executed during the compile-time.

But how is the compiler going to load `forAllMatches`, if this function hasn't been compiled yet?

It's theoretically possible to imagine a compiler that automatically analyzes the dependencies, compiles macros first and only then proceeds with the normal code, however our compiler isn't (yet) capable of that.

Thus for the moment macros have to be compiled separately from the rest of the application. Also one has to remember to add macro classes to the library classpath (`-cp`). On these slides, I don't specify the paths explicitly, because my CLASSPATH environment variable is configured to contain current directory.

First results

```
> scalac -Xmacros Rx.scala
> scalac -Xmacros Test.scala
Test.scala:5: error: exception during macro expansion:
scala.NotImplementedError: an implementation is missing
  at scala.Predef$.$qmark$qmark$qmark(Predef.scala:233)
  at Mx$.defmacro$forAllMatches(Rx.scala:37)
  at java.lang.reflect.Method.invoke(Method.java:597)
  at scala.reflect.runtime.Mirror.invoke(Mirror.scala:46)
  at ...Macros$class.macroExpand(Macros.scala:135)
```

Much better! The macro is invoked - and, of course, crashes with `NotImplementedError`.

Note that this exception doesn't blow the compiler up, but rather gets processed as a normal type erorr (one can even see the position that points to a macro call that has caused the error).

Creating the trees

Let's proceed with implementing the meat of the macro. As we've seen, macros work on the level of abstract syntax trees, so macro arguments will be ASTs and we'll have to generate an AST as a return value. This is scary - the exact format of ASTs is not documented, so what should we do?

We'll make good use of built-in logging and prettyprinting facilities of `scalac`.

On the one hand, `Tree` (the base class of Scala AST nodes) has `toString` overriden to print its contents as a nicely formatted code in Scala. On the other hand, with the help of `showRaw` one can get a detailed inspection of the structure of an input AST.

This functionality is also available via `-Xshow-phases` and `-Yshow-trees` compiler flags.

Parameters of the macro

Let's start the investigation from studying the parameters that get passed to the macro. To do that, let's inspect the exact shape of macro's binary representation.

One approach to that is looking into Macros.scala. Another approach involves running compilation of the macro with `-Ydebug`, which will print the desugaring of the macro:

```
> scalac -Xmacros -Ydebug Rx.scala
[running phase typer on Rx.scala]
macro def: def defmacro$forAllMatches
      (glob: api.this.Universe)
      (_this: glob.Tree)
      (pattern: glob.Tree, f: glob.Tree): glob.Tree = {
  implicit val $glob: glob.type = glob;
  import $glob._;
  $qmark$qmark$qmark
}
added to module class Mx: method defmacro$forAllMatches
```

Parameters of the macro

First things first. Our macro gets the compiler context - the `glob` of type `Universe` (public aspect of the compiler API that exposes the most important functionality, e.g. trees and types).

Secondly (I hope, this is old news by now), macro parameters come into a macro in the form of abstract syntax trees. This includes `pattern`, `f` and even `_this` (our macro gets called as an instance method, so it has to have its `this`).

Another interesting point - both the parameters and the return value have path-dependent types that are parameterized with the context. This is done on purpose - most of compiler data structures (including trees) are mutable, so we need to prevent their sharing between different compiler instances.

Inspecting parameters

Finally, we can take a look at what exactly gets passed in macro parameters. To do that, we'll use good old tracing and insert a couple of `showRaw`'s in the body of the macro.

```
_this: Apply(Select(Ident(newTermName("Rx")),
    newTermName("string2mx")),
    List(Select(This(newTypeName("Test")),
    newTermName("s"))))

pattern: Literal(Constant("..."))

f: Apply(Ident(newTermName("println")),
    List(Apply(Select(Literal(Constant("...")),
    newTermName("format")),
    List(Ident(newTermName("key")),
    Ident(newTermName("value"))))))
```

Generation of the result

After getting the idea about the shape of the input parameters, let's decide on what exactly the macro will be expanded into. I like the following template:

```
"...".findAllIn(s).matchData.foreach(m => {
  val key = m.group("key")
  val value = m.group("value")
  println("...".format(key, value))
}))
```

To find out what trees we need to emit the required code, we can use the aforementioned trick with `-Xprint:parser`. Flag `-Yshow-trees` will almost verbatimly show the code that is necessary to hand-craft the trees.

Notes on `-Yshow-trees`

`-Yshow-trees` is very useful, but quite often one has to apply mental postprocessing to its results.

Firstly, the names get printer as regular string, but the compiler represents them with the objects of type `Name`. Consequently, printed strings have to be wrapped in `newTermName` or `newTypeName` (depending on the context).

Secondly, lists of objects get marked only by indentation, whereas the `List(...)` constructor doesn't get printed at all. To find out what exactly has to be wrapped in lists, one has to consult the declarations of `Tree` subclasses (link to Trees.scala).

Finally, modifiers get dumped in a not very intuitive way. We won't discuss this point here and will just mention that `Modifiers()` creates an empty modifiers object, which is sufficient for the purposes of the demo.

PROFIT!

Having acquired all the aforementioned knowledge, we can implement the body of the macro and get the long-awaited result (or we can cheat and peek into Rx.scala in the `result` branch).

```
> scalac -Xmacros Rx.scala
> scalac -Xmacros Test.scala
> scala Test
key = foo, value = bar
```

Final question. Try to predict what will happen, if you move the regex literal into a local variable, i.e. if you write as follows:

```
val s = "foo=bar"
val pattern = """^(?<key>.*?)=(?<value>.*)$"""
s.forAllMatches(pattern, println(...))
```

Outline

Quasiquotes

During the talk I was asked a question. "Manual assembling of trees is incovenient. However the parser is already capable of transforming Scala code into trees. Why not take advantage of this to create trees using their string representation?".

And indeed, creation of ASTs can be simplified using the mechanism known as "quasiquoting".

I originally intended to avoid this topic, because the design of quasiquotes (in contrast with out vision of macros) is by far not settled down, so the final version might radically differ from the current prototype.

However, the functionality provided by quasiquotes is truly important in the context of our discussion, so we'll cover this topic over the course of the next slides. More information about quasiquotes is available in my previous talk

What are quasiquotes capable of?

Quasiquotes turn strings with code into equivalent abstract syntax trees:

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

Smaller quasiquotes can be inserted into bigger ones by the virtue of splicing:

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

Quasiquotes can be used as patterns to match the structure of syntax trees and bind the parts of ASTs to variables:

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

Generalized quasiquotes

Quasiquoting mechanism can be generalized to support embedding of completely alien domain-specific languages.

By implementing parsing, splicing (and, possibly, pattern matching) for a DSL, the programmer ensures tight integration of a language into Scala. Later this facility can be used by simply prepending the corresponding quasiquote id to a string literal that contains a program in a DSL.

Quasiquote provider can be run during the compile-time, so it can use semantic information about the program (symbol tables, types, etc) and, if necessary, perform precompilation (similarly to what we have recently done for regexes).

Fancy ways of leveraging quasiquotes to create domain-specific languages are covered in a paper about Template Haskell: Why It's Nice to be Quoted: Quasiquoting for Haskell.

Outline

Summary

- Macros provide transparent extension points into the compiler. By the virtue of macros, one can analyze and optimize code, generate boilerplate and much more.
- Macros in Scala are no longer a dream. In the foreseeable future we will release a beta version and present a SIP for adding macros to 2.10.
- Of a significant interest to us are quasiquotes. They not only simplify working with macros, but also enable Scala to subsume external DSLs. This feature is in its early stage of development, but we'll work hard to implement it by 2.10.
- Right now you can follow these slides to experiment with your own macros. If something doesn't work out - please, drop me a line at eugene.burmako@epfl.ch, and we'll have it sorted.

What next?

Out next steps are stabilizing reflection (that is used by macros to expose ASTs and the compiler to the developer) and releasing a beta version of macros.

Along with the beta version we will publish the documents within Scala Improvement Process and will suggest macros and, possibly, quasiquotes for being included in 2.10 (that is scheduled to be released within a few months).

You can stay tuned by following: news at scalamacros.org (official announcements), posts in my LJ (design notes and milestones), mt twitter messages (unstructured ramblings :)).

Finally, you can watch our `scala/scala` fork at GitHub: https://github.com/scalamacros/kepler. Stable stuff gets accumulated in `develop`, experimental features (e.g. quasiquotes) have their own branches.

Questions and answers

eugene.burmako@epfl.ch