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

Постановка задачи

В Скале существует несколько способов использования регексов для нахождения фрагментов текста, удовлетворяющих определенным условиям.

Во-первых, это использование императивного `Match.group`, по старинке. Во-вторых, можно воспользоваться декларативным паттерн-матчингом при помощи `Regex.unapply`. Однако оба подхода имеют один и тот же недостаток. При использовании в регексах именованных групп имена придется писать дважды - один раз в регексе и второй раз при обработке матча.

Хочется, чтобы имена групп в регексе автоматически маппились на переменные в лексическом скоупе, то есть, чтобы можно было сделать вот так:

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

Интерактивная презентация

За ходом реализации макроса можно проследить вживую. Для этого клонируем https://github.com/xeno-by/alphakeplerdemo и выбираем бранч `skeleton`.

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

Скелет содержит всю необходимую обвязку, оставляя нереализованным лишь тело макроса. Исходники надо компилировать именно компилятором, который лежит рядом, так как они используют фичи, которых еще нет в апстриме.

И еще одно замечание. Java до версии 1.7 не поддерживает именованные группы внутри макросов, поэтому я написал небольшой воркараунд. Этот подход не годится для продакшена (неправильно работает с вложенными группами), но вполне неплох для демки.

Знакомимся со скелетом макроса

Откроем Rx.scala, в котором уже определен макрос в виде икстеншн-метода (да, так можно делать!) с телом, содержащим три вопросика (вы знали о такое фиче?):

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

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

Сигнатура макроса

В сигнатуре макроса нет ничего необычного, кроме одного момента - подчеркивания вместо типа параметра `f`.

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

Подчеркивание обозначает тот факт, что параметр не будет типизироваться тайпером, а сразу будет передан в макрос (эта нотация весьма свежая, поэтому есть шанс, что к релизу она изменится, но пока что будем использовать именно ее).

Но зачем это надо? Почему бы не указать тип `f: => Unit`, аналогично тому, как это сделано в обычном `foreach`?

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

Как раскрываются макросы

Обычная последовательность обработки макроса аналогична таковой для обработки вызова функции: вначале типизируются аргументы, после чего раскрывается макрос (что такое раскрытие макроса? это вызов его тела с набором AST, которые соответствуют его параметрам).

Рассмотрим этот механизм на примере нашего гипотетического макроса:

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

С паттерном все понятно - он будет типизирован как обычная строка, но что делать с функцией? Можем ли мы ее типизировать?

Типизация аргументов макросов

На самом деле мы не можем типизировать выражение с `println` потому, что оно использует переменные `foo` и `bar`, которых еще нет в лексическом окружении - они будут магическим образом сгенерированы макросом.

Конечно, в нашем случае живой человек мог бы догадаться, что эти переменные имеют тип `String`, но для компилятора макрос абсолютно непрозрачен (так как он выполняется уже после типизации аргументов). Возможно, в зависимости от фазы луны макрос будет возвращать то строки, то инты - что тогда делать?

Поэтому в данном случае компилятор вынужден пропустить типизацию выражения, передаваемого в макрос. Означает ли это, что макросы отменяют статическую проверку типов?

Типобезопасность макросов

Со статической проверкой типов все в порядке, так как AST, которое вернет макрос в результате раскрытия, будет сразу же тайпчекнуто.

Для интереса вот фрагмент из Typers.scala, который демонстрирует эту логику.

```
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)
}
```

Временный отказ от типизации неприятен - в худшем случае ошибки в аргументах макроса будут обработаны позже, чем следовало бы, и в другом контексте, что может привести к путанице. Но эта жертва необходима для должной гибкости (см. ретроспективу по Template Haskell насчет деталей).

Тестовый запуск

Теперь, когда мы разобрались с точкой отсчета, попробуем запустить код как есть и посмотреть что случится. Наверняка, будет кое-что интересное.

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

Упс! Результат занимательный, но не очень вдохновляющий. Мы наткнулись на проблему совместной компиляции макросов и их клиентов.

Раздельная компиляция

Дело вот в чем. Как упоминалось раньше, макросы - это мини-плагины к компилятору, которые загружаются из classpath и выполняются во время компиляции.

Но как компилятор загрузит `forAllMatches`, если тот еще не скомпилировался?

Можно теоретически представить себе компилятор, который автоматически анализирует зависимости и вначале компилирует макросы, после чего принимается за обычный код, но наш компилятор (пока что) так не умеет.

Поэтому на текущий момент макросы надо компилировать отдельно от основной программы. При этом для компиляции программы надо не забыть добавить классы макросов в пользовательский classpath (`-cp`). На этих слайдах я не указываю путь к макросам явно, так как моя переменная окружения CLASSPATH содержит текущую папку.

Первые результаты

```
> 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)
```

Уже лучше! Макрос действительно вызывается - и падает с `NotImplementedError`.

Заметим, что это исключение не рушит компилятор, а обрабатывается как обычная ошибка типизации (можно даже увидеть позицию, указывающую на макрос, при раскрытии которого произошла ошибка).

Создаем деревья

Теперь приступим к написанию логики макроса. Как уже упоминалось, макросы работают на уровне абстрактных синтаксических деревьев, поэтому на входе у нас будут AST и на выходе мы тоже должны сгенерировать AST. Это весьма пугающий факт - формат AST нигде не документирован, что же нам делать?

Нам помогут встроенные средства scalac по логгированию и преттипринтингу своих структур данных.

С одной стороны, у `Tree` (базового класса нодов AST Скалы) перегружен метод `toString`, которые печатает деревья в виде симпатично отформатированного кода на Скале. С другой стороны, при помощи `showRaw` можно детально проинспектировать структуру AST.

Эта функциональность также доступна через такие флаги компилятора как `-Xshow-phases` и `-Yshow-trees`.

Параметры макроса

Начнем исследование с изучения того, какие параметры приходят в макрос. Для этого определимся, во что именно макрос компилируется.

Для этого можно посмотреть в файл Macros.scala, а можно запустить компиляцию макроса с флагом -Ydebug, в результате чего распечатается развернутый макрос:

```
> 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
```

Параметры макроса

Начнем по порядку. Во-первых, в макрос передается контекст компилятора - переменная `glob` типа `Universe` (публичный срез API компилятора, который содержит самые важные фичи вроде деревьев и типов).

Во-вторых (надеюсь, к этому моменту это уже не новость), параметры макроса приходят в виде AST: `pattern`, `f` и даже `_this` (макрос вызывается как инстанс метод, поэтому у него должен быть `this`).

Еще один интересный момент - и параметры, и возвращаемое значение имеют path-dependent типы, которые зависят от контекста. Это сделано неслучайно - деревья компилятора мутабельные, поэтому нужно предотвратить их расшаривание между разными экземплярами компиляторов.

Исследование параметров

Наконец, можно посмотреть, что же именно приходит в параметры макроса. Для этого воспользуемся проверенным методом - вставим в тело макроса печать деревьев при помощи showRaw.

```
_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"))))))
```

Генерация результата

После того, как мы получили представление о входных параметрах, давайте решим, во что именно будет раскрываться макрос. Мне нравится вот такой код:

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

Чтобы определить, какие деревья нужны для того, чтобы сгенерировать требуемый код, можно воспользоваться упомянутым выше трюком с `-Xprint:parser`. Флаг `-Yshow-trees` почти дословно покажет код, необходимый для создания деревьев вручную.

Замечание о работе `-Yshow-trees`

`-Yshow-trees` очень полезен, но к нему зачастую приходится применять ментальный постпроцессинг.

Во-первых, имена дампер печатает в виде обычных строк, но в компиляторе они представляются объектами типа `Name`. Соответственно, эти строки надо оборачивать в `newTermName` или `newTypeName` в зависимости от контекста.

Во-вторых, списки объектов выделяются отступами, а конструктор `List(...)` на печать не выводится. Чтобы узнать, что именно необходимо оборачивать в списки, лучше всего проконсультироваться с объявлениями подклассов `Tree` (ссылка на Trees.scala).

Наконец, модификаторы выводятся в не очень наглядном виде. Мы не будем подробно останавливаться на этом моменте, только отметим, что `Modifiers()` создает пустой объект-модификатор.

PROFIT!

Вооружившись полученными знаниями и терпением, можно дописать макрос и получить долгожданный результат (подсказка находится в файлике Rx.scala бранча `result`).

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

Вопрос на засыпку. Попробуйте предсказать, что получится, если строчку-регекс вынести во временную переменную, то есть написать вот так:

```
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