α-Кеплер

Евгений Бурмако

EPFL, LAMP

14 января 2012

α-Кеплер

Всем привет! Меня зовут Евгений Бурмако, с прошлой осени я работаю в Scala Team и параллельно учусь в аспирантуре EPFL на кафедре Мартина Одерского.

Сегодня мы обсудим прогресс проекта “Кеплер”, в рамках которого разрабатываются **макросы** и **квазицитаты** - средства метапрограммирования времени компиляции для Скалы.

В предыдущем выступлении я рассматривал макросы с теоретической точки зрения, а сегодня будет, в основном, практика. Поэтому перед тем, как продолжить чтение, может быть полезно просмотреть слайды прошлого рассказа.

После небольшого введения мы реализуем одну штуку, о которой я мечтал со времен знакомства с Немерле - конечно же, в виде макроса. Также в плане обсуждение абстрактных синтаксических деревьев Скалы в контексте разработки макросов. Приятного чтения!

План

Начнем издалека

Одним из интересных событий этой осени в мире Скалы стало обсуждение интерполяции строк.

Эта функциональность встречается во многих скриптовых языках (bash, Perl, Ruby) и предоставляет возможность встраивать в строки переменные из лексического окружения.

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

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

Буква s, стоящая прямо перед строкой - не опечатка, а обозначение того, что строка интерполируется (в целях обратной совместимости доллары в обычных строках не будут интерпретироваться специальным образом).

Реализация интерполятора

Несмотря на свою простоту, интерполяция не поддается библиотечной реализации, так как она оперирует лексическим окружением, которое в явном виде недоступно.

В рамках смелого научного эксперимента нам придется открыть `doTypedApply` (функцию, которая типизирует применение методов) и вставить проверку на метод интерполяции. Внутри компилятора у нас есть полный доступ ко всей семантической информации о программе, чем мы и воспользуемся.

В тайпере возможно получить строковый литерал, который требуется проинтерполировать, и прямо на месте распарсить этот литерал. После этого, пройдясь по дереву контекстов, несложно составить словарь видимых переменных и превратить вызов метода `s` в обычную конкатенацию строк.

Если очень интересно, детали реализации можно посмотреть у меня в репозитории, а пока что пойдем дальше.

Использование интерполятора

Пока что все было довольно несложно. Мы встроились в тайпер и заменили вызов функции-маркера на вручную собранное дерево. Звучит страшнее, чем оно есть на самом деле. Через несколько слайдов мы будем заниматься тем же самым в режиме лайв =)

Единственный вопрос в том, как использовать наш патч к компилятору. Интерполяция выглядит более-менее серьезно, но вот трюки, специфичные для проекта, в апстрим явно не примут, а таскать за собой кастомную сборку компилятора неудобно по многим причинам.

Общепринятым решением в данной ситуации является написание плагина к компилятору (например, CPS в Скале реализован именно через плагин), а это очень близко к философии макросов. Мы почти на месте.

Что такое макросы?

Макросы - специального вида плагины к компилятору, которые автоматически загружаются из classpath и преобразуют заданные элементы программы (вызовы функций, ссылки на типы, объявления классов и методов).

По аналогии с рассмотренным выше примером макросы:

- Выполняются во время компиляции
- В отличие от традиционных плагинов прозрачно загружаются компилятором, не требуя оборачивания в `Plugin` и `Component`
- Работают с деревьями выражений, которые соответствуют коду компилируемой программы
- Имеют доступ к внутренним сервисам компилятора (рефлексия ранее скомпилированного кода, вывод и проверка типов, лексические окружения и так далее)

Какие бывают макросы?

Макро-функции получают на вход AST аргументов, раскрываются в AST и инлайнятся в точку вызова.

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

Макро-типы представляют собой типы, члены которых генерируются во время компиляции:

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

Макро-аннотации выполняют пост-обработку объявлений методов и типов:

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

Отклик

За несколько месяцев жизни проекта “Кеплер” нашлось немало практических задач, которые сильно упрощаются при наличии макросов.

Уже есть желающие использовать макросы для реализации языка запросов в O/RM, я слышал о планах использовать макро-аннотации в линзах, есть идеи насчет применения макро-типов для генерации бойлерплейта, необходимого для вычислений на типах. Буквально позавчера Мартин реализовал прототип оптимизатора, который использует макросы для ускорения `Range.foreach`.

Наша лаборатория выиграла грант Швейцарской комиссии по технологиям и инновациям на разработку усовершенствования Скалы, в основе которого лежат рефлексия и макросы.

Статус

На сегодняшний момент мы переосмыслили публичный интерфейс к компилятору и его структурам данных и воплотили в жизнь прототип компилятора, который реализует макро-функции. Мы довольны тем, как выглядит прототип, и будем дальше двигаться в этом направлении.

В активной разработке находятся квазицитаты - доменно-специфический язык для создания и декомпозиции абстрактных синтаксических деревьев. Скорее всего, они будут реализованы как обобщение строковой интерполяции, но здесь гораздо меньше ясности, чем с макросами.

Мы опубликовали сайт scalamacros.org, на котором собраны материалы по нашему проекту. Особый интерес, на наш взгляд, составляет раздел “Use Cases”, в котором рассматриваются области применения макросов.

План

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(...))
```

План

Квазицитаты

В процессе демонстрации был задан вопрос. "Собирать деревья вручную неудобно. Но ведь парсер компилятора умеет преобразовывать код на Скале в деревья. Почему бы не создавать деревья, пользуясь их строковым представлением?"

И действительно, создание AST можно упростить с помощью механизма, известного как "квазицитирование".

Я хотел обойти эту тему стороной потому, что дизайн квазицитат (в отличие от макросов) даже близко не завершен, поэтому финальная версия может радикально отличаться от текущего прототипа.

Однако предоставляемый квазицитатами функционал действительно очень важен, поэтому мы немного поговорим на эту тему на следующих слайдах. Более подробно квазицитаты разобраны в предыдущем выступлении.

Что умеют квазицитаты?

Квазицитаты превращают строки с кодом в эквивалентные синтаксические деревья:

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

Маленькие квазицитаты можно вставлять в большие при помощи сплайсинга:

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

Квазицитаты можно использовать слева от стрелочки при паттерн-матчинге:

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

Обобщенные квазицитаты

Механизм квазицитирования можно обобщить для встраивания совершенно чужеродных доменно-специфических языков.

Реализовав парсинг, сплайсинг (и, возможно, паттерн-матчинг) для DSL, программист обеспечивает тесную интеграцию языка с Scala. Для использования провайдера необходимо просто задать идентификатор провайдера перед строчкой с квазицитатой.

Провайдер квазицитаты выполняется во время компиляции, поэтому он может использовать семантическую информацию о программе (таблицы символов, сервис проверки типов, и т.д.), а также выполнять прекомпиляцию (аналогично тому, что мы недавно делали для регексов).

Интересные способы применения квазицитат для создания доменно-специфических языков рассматриваются в работе Why It's Nice to be Quoted: Quasiquoting for Haskell.

План

Резюме

- Механизм макросов предоставляет прозрачные точки расширения компилятора. С помощью макросов можно анализировать и оптимизировать код, генерировать бойлерплейт и многое другое.
- Макросы в Скале - это уже реальность. В ближайшем будущем мы выпустим бета-версию и представим пропоузал по добавлению макросов в 2.10.
- Интересное направление исследования - квазицитаты. Они не только упрощают работу с макросами, но и позволяют интегрировать в Скалу внешние DSL. Эта фича находится на более ранней стадии по сравнению с макросами, но мы постараемся успеть сделать ее к 2.10.
- По шагам нашей демонстрации уже сейчас можно поэкспериментировать со своими собственными макросами. Если что-то не получится - напишите мне на eugene.burmako@epfl.ch, разберемся вместе.

Что дальше?

Наш следующий шаг - стабилизация рефлексии (которая используется макросами для доступа к деревьям и компилятору) и выпуск бета-версии макросов.

Вместе с бета-версией мы опубликуем документы в рамках Scala Improvement Process, в которых предложим включить макросы и, возможно, квазицитаты в версию 2.10 (которая выйдет в первой половине 2012 года).

За новостями можно следить по следующим направлениям: новости на scalamacros.org (официальные объявления), посты в моем ЖЖ (дизайн-заметки и майлстоуны), мессаги в моем твиттере (с этим все понятно :)).

Наконец, за развитием проекта можно наблюдать на гитхабе: https://github.com/scalamacros/kepler. Основная разработка ведется в `develop`, экспериментальные вещи (вроде квазицитат) имеют собственные бранчи.

Вопросы и ответы

eugene.burmako@epfl.ch