Проект “Кеплер”: макросы для Скалы

Евгений Бурмако

EPFL, LAMP

29 октября 2011

План

Проект “Кеплер”

Цель: Реализовать макросы, типобезопасный механизм метапрограммирования времени компиляции для Скалы.

Идеи: В основном из Немерле, хотя будет интересно поэкспериментировать с некоторыми замечательными моментами из Scheme, MetaML и Template Haskell.

Участники: Евгений Бурмако (ваш покорный слуга), Кристофер Фогт (автор Scala Integrated Query), Штефан Цайгер (автор ScalaQuery).

Статус: В процессе написания спецификации Scala Improvement Proposal, некоторые аспекты уже обсуждаются в рамках SIP “String interpolation and formatting”.

В двух словах

Уже очень долгое время макросы в популярных языках программирования ассоциируются с макросами C/C++. Неудивительно, что многие автоматически воспринимают слово “макрос” в штыки.

В отличие от макросов препроцессора, наши макросы:

- Представляют собой код на полноценной Скале
- Работают с высокоуровневыми и типизированными деревьями выражений
- Выполняются в контексте компилятора, поэтому имеют доступ ко всей семантической информации, доступной компилятору
- Не изменяют синтаксис Скалы

Наши макросы напоминают макросы Лиспа, доработанные для поддержки богатого синтаксиса и статической типизации.

Прежде, чем мы начнем

Мне крайне интересно услышать ваши комментарии, предложения и критику. Дизайн макросов только-только начал обретать твердые очертания, поэтому его можно подточить для поддержки ваших проектов и сценариев использования.

За всего лишь месяц жизни проекта уже было пару случаев, когда я вносил в черновик спецификации значительные изменения для поддержки тех или иных сценариев, о которых я узнавал в процессе дискуссий. Пожалуйста, не стесняйтесь и пишите мне на eugene.burmako@epfl.ch.

Кроме того, все нижеописанное является крайне экспериментальным и может быть в любой момент изменено на 180°. На текущий момент проект “Кеплер” *просто* является моей будущей диссертацией, поэтому ни Scala Solutions, ни TypeSafe никак с ним не связаны.

План

Макросы на микро-уровне

```
macro printf(format : string, params parms : array[expr])
{
  def (evals, refs) = make_exprs(parse(format), parms);
  def seq = evals + refs.Map(x => <[ Write($x) ]>);
  <[ { ..$seq } ]>
}

printf("Value = %d", 123 + 877)

{
  def _N_1812 = (123 + 877 : int);
  Console.Write("Value = ");
  Console.Write(_N_1812)
}
```

Макросы

Это был довольно сложный фрагмент кода (кстати, код этот написан на Немерле). Давайте подробно рассмотрим концепции, относящиеся к макросам:

- `Printf` - макрос. Его тело представляет собой метакод, т.е. код, который исполняется во время компиляции.
- В качестве аргументов `printf` принимает набор выражений времени компиляции, т.е. узлов AST. Возвращаемое значение тоже является AST.
- Когда компилятор встречает вызов макроса, он исполняет метакод (т.е. тело макроса) и вставляет *[splice]* его результат (т.е. сгенерированный код) в место вызова.
- Несмотря на то, что макросы исполняются на этапе компиляции и не имеют доступа к значениям времени выполнения, они могут вызывать произвольные функции и методы, в том числе и стандартную библиотеку.

Квазицитаты

Генерация кода обычно весьма нудное занятие, но в рассматриваемом примере нам удалось воспользоваться весьма элегантной техникой:

- Прикольные скобки `<[ ... ]>` задают квазицитаты *[quasiquotations]*, фрагменты кода, которые можно удобно композировать и декомпозировать.
- Большие фрагменты кода можно собирать из маленьких при помощи так называемого сплайсинга *[splicing]*. Выражения `$x` и `..$seq` являются сплайсами.
- Квазицитаты возвращают значения времени компиляции. Отметим важный факт, который резюмирует все, что мы на текущий момент знаем про макросы: возвращаемое значение макроса из примера (последняя строчка в теле макроса) - квазицитата, т.е. AST. Все сходится!

Сопоставление с образцом

Квазицитаты также находят применение в сопоставлении с образцом для удобной декомпозиции абстрактных синтаксических деревьев. Например, некий гипотетический оптимизатор может выглядеть следующим образом:

```
match (e) {
    ...
    | <[ $x * 1 ]> => <[ $x ]>
}
```

Будучи использованными с левой стороны сопоставления с образцом, сплайсы задают именованные “отверстия” в квазицитатах.

Очень красивый трюк, правда? Но он не исчерпывает феерию синтаксических чудес на сегодня. О еще более захватывающих вещах можно почитать в моем блоге.

Гигиена

За кадром остались детали разбора строки формата и сборки кусочков выходной строки. Во всех подробностях реализацию можно посмотреть в статье Влада Чистякова, но пока что мы остановимся на одном интересном фрагменте.

Для того, чтобы в сгенерированном коде сослаться на переменные, доступные в месте вызова, в метакоде написано `<[ $(expr : usesite) ]>`. Зачем все так усложнять?

Все это нужно потому, что макросы Немерле являются гигиеничными, т.е. они переименовывают генерируемые переменные и ссылки на переменные для того, чтобы предотвратить случайные пересечения имен.

Чаще всего это как раз то, что нужно, поэтому гигиена включена по умолчанию, но иногда (например, для реализации строковой интерполяции) это мешает, поэтому при желании гигиену можно обойти.

Макросы на макро-уровне

```
[Usage(Phase.BeforeInheritance, Targets.Class)]
macro Serializable (t : TypeBuilder)
{
  t.AddImplementedInterface (<[ ISerializable ]>)
}

[Serializable]
class S {
  public this (v : int, m : S) { a = v; my = m; }
  my : S;
  a : int;
}
```

Кодогенерация

Макросы верхнего уровня, которые применяются путем аннотирования сущностей программы, являются ответом Немерле на проблему дублирования кода.

Такие макро-аннотации крайне полезны для устранения копипасты, против которой бессильны ООП и ФП. Особенно приятно использовать подобные макросы тем, кто уже имеет опыт работы с текстовыми генераторами вроде T4 - ведь в данном случае все делается через квазицитаты, которые являются высокоуровневыми, типобезопасными и композируемыми.

При соответствующей поддержке со стороны языка в квазицитатах можно задавать произвольные целевые языки (например, HTML), что делает макросы еще более соблазнительными.

Средства разработки

Макросы содержат в себе немало магии, поэтому для работы с ними требуется надежная поддержка средств разработки:

- В силу того, что макросы изменяют порядок сборки (перед компиляцией основной программы, очевидно, необходимо скомпилировать используемые в программе макросы), средства сборки должны быть в курсе макросов.
- Генерируемый код может понадобиться оттрассировать в отладчике. Отсутствие исходников не является существенной проблемой, ибо мы можем сгенерировать их из AST, но этот вопрос все равно надо обдумать.
- Для того, чтобы обеспечивать правильную подсветку и анализ кода, IDE должны знать о макросах, причем макросы также должны знать об IDE и вести честную игру - быть реентерабельными, быть аккуратными с разделяемым состоянием.

Резюме

- В отличие от печально известных макросов препроцессора C/C++, современные макросы работают с AST и тесно интегрированы в язык и компилятор, что делает их безопасными, выразительными и композируемыми.
- Макросы исполняются на этапе компиляции, но при этом имеют доступ к рантайму и библиотекам языка. Таким образом, например, во время компиляции макрос может обратиться к базе данных.
- Микро-уровень макросов предоставляет средства для виртуализации языка программирования и создания доменно-специфических языков. Макро-уровень позволяет автоматически сгенерировать части программы.
- (Фанатам ASM и `Reflection.Emit` посвящается) Квазицитаты делают анализ и генерацию кода приятным и продуктивным занятием.

План

На микро-уровне

- Разработка встроенных доменно-специфических языков. Одним из паттернов eDSL является перегрузка операций языка и протаскивание действий пользователя через весь DSL. Макросы позволяют избежать этой весьма хрупкой процедуры, так как они могут получить информацию о замысле пользователя непосредственно из AST.
- Синтаксический сахар для монад. Важным частным случаем языковой виртуализации является do-нотация для монад. С помощью макросов ее можно реализовать без поддержки со стороны компилятора.
- Встраивание внешних доменно-специфических языков. При помощи квазицитат в программу можно встраивать фрагменты кода на произвольных языках. Макросы-обработчики квазицитат могут обеспечить интеграцию этих языков с главным языком программирования (общий лексический контекст, услуги типизации и т.п.).

На макро-уровне

- Генерация кода любых разновидностей. С точки зрения кодогенерации макросы верхнего уровня более продуктивны, чем текстовые генераторы, и гораздо более высокоуровневы, чем генераторы байт-кода.
- Аспектно-ориентированное программирование. Традиционные парадигмы не способны справиться с некоторыми задачами декомпозиции и дедубликации кода. В результате возникли частные подходы к решению таких задач. Макросы предоставляют средства для склеивания аспектов приложения в общем случае.
- Типы данных “a la carte”. Часто бывает необходимо собрать класс из нескольких независимых кусочков. Для этого очень хороши трейты, но иногда их гибкости не хватает, например, для кастомизации case-классов.

Практический пример: LINQ

```
var db = new MyDbContext(connString);
var products = db.Products;
products.Where(p => p.Name.StartsWith("foo")).ToList()
```

Переменная `products` в этом примере имеет тип, унаследованный от `IQueryable<Product>`.

Метод `Where` доступен для всех коллекций стандартной библиотеки .NET, но для `IQueryable` он перегружен магическим способом, который говорит компилятору передать в него не функцию-фильтр, а ее представление в виде дерева выражений.

В процессе жизненного цикла запрос накапливает в себе информацию о вызванных методах типа `Where` и переданных в виде деревьев параметрах, после чего по просьбе программиста (`foreach`, `ToList` и так далее) преобразует эту информацию в запрос к конкретному источнику данных.

Постановка задачи

LINQ - неувядаемая классика нашего времени, так сказать, де-факто стандарт в области виртуализации запросов в мейнстримных языках программирования, поэтому хочется иметь что-то подобное в стандартной библиотеке Скалы.

В следующем разделе презентации мы увидим как при помощи макросов можно реализовать не только LINQ, но и кое-что поинтереснее - причем все это исключительно средствами языка, т.е. без внесения изменений в компилятор.

Кроме независимости от компилятора полученное решение исправит некоторые фундаментальные ограничения LINQ, например, проблемы с композируемостью - все это благодаря строгости и ортогональности концепции макросов.

План

LINQ на макросах

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

val products = db.products
products.filter(p => p.Name.startsWith("foo")).toList
```

Макро-определения

Как можно заметить, вызов макроса `filter` в предыдущем примере неотличим от вызова обычного метода. Это неудивительно, ведь `macro def` читается как “макрос, который может использоваться везде, где бы использовался `def`”.

Сигнатура макроса также прозрачно интегрирована в целевой язык программирования. Макросы исполняются во время компиляции, поэтому они принимают на вход и возвращают исключительно нетипизированные AST. Однако спецификации типов позволяют компилятору проверить корректность макроса и как можно раньше сообщить о возможных ошибках.

Кроме того, внутри макроса у программиста есть доступ ко всем переменным, которые составляют область видимости, в том числе к `this` и к неявным переменным. Для внимательных: да, в макросах перегружен метод `implicitly[T]`.

Легковесные, но не убогие

Из-за своей легковесности макросы могут казаться ограниченным хаком, но это не так. На самом деле, макросы - полноценные граждане языка и рантайма.

С одной строны, легковесность является синтаксическим сахаром, за которым скрывается обычная функция, которая принимает на вход несколько AST и возвращает AST на выход. С другой стороны, в процессе компиляции макросы преобразуются в обычный байткод, который может быть использован при сборке другой единицы компиляции.

Наконец, макросы могут использовать произвольные функции/методы и загружать любые библиотеки. За примером далеко ходить не надо - `reify` из предыдущего листинга является обычной функцией `AST` $\Rightarrow$ `AST`. Кстати, а что будет, если вместо `${reify(p)}` написать `reify($p)`? Подсказку посмотреть в черновике спецификации квазицитат.

Тайп провайдеры

Языковая поддержка интегрированных запросов и фреймворк трансляции запросов хороши как вещи в себе, но кроме них еще необходим способ отображения сущностей источника данных на классы нашего языка программирования.

Это хорошо известная проблема в мире O/RM. Один из путей ее решения заключается в использовании текстовой кодогенерации, другой - в ручном кодировании классов. Сейчас мы рассмотрим альтернативный подход, который стремительно набирает известность: тайп провайдеры, разработанные в Microsoft Research.

Идея тайп провайдеров заключается в том, чтобы во время компиляции генерировать классы для сущностей источника данных (таблиц, сущностей XSD, записей LDAP). Если мы слышим “время компиляции”, значит, можно применить макросы.

Тайп провайдеры на макросах

```
macro class MySqlDb(connString: String) = ...
type MyDb = MySqlDb("Server=127.0.0.1;Database=Foo;")

val products = new MyDb().products
products.filter(p => p.Name.startsWith("foo")).toList
```

Макро-типы

. . . это не более, чем синтаксический сахар для старых добрых макросов, преобразующих AST в AST.

Этот вид макросов также принимает на вход AST и также возвращает AST, но возвращаемое выражение является не просто выражением языка, а полноценным определением класса. Здесь нет двойного дна - все действительно настолько просто.

В силу того, что макросы являются полноценными сущностями языка, разработчики могут повторно использовать и частично переопределять логику уже существующих макро-типов.

Также, благодаря прозрачности макросов, пользователи могут наследоваться от макро-типов и переопределять их функциональность в терминах объектно-ориентированной парадигмы.

Минимализм

Вначале я хотел развить идею macro чего-то и дальше и поэтому обдумывал разные варианты: например, macro package или macro annotation. Мартин был сильно против раздувания языка, и это заставило меня задуматься. Выяснилось, что можно обойтись только определениями и типами.

Скажем, для генерации большого количества классов (например, `TupleN`) можно написать `macro object` и потом импортировать его в место использования. Для генерации классов в текущий пакет можно подмешать `macro trait` в `package object`.

Специальные макро-аннотации тоже имеют сомнительную ценность, ибо их можно заменить обычными аннотациями, а раскрытие макросов можно запускать, подмешивая в объявляющий класс специальный `macro trait`.

Резюме

- Наши макросы прозрачно интегрированы в язык программирования. Макро-определения (`macro defs`) практически неотличимы от обычных определений, макро-типы (`macro types`) могут быть использованы почти везде, где используются обычные типы.
- С помощью этих двух конструкций языка можно эффективно реализовать анализ кода, языковую виртуализацию, а также победить некоторые виды дубликации кода, с которыми не справляются традиционные парадигмы программирования.
- Наконец, макро-типы вкупе в механизмом квазицитирования предоставляют высокоуровневый интерфейс к кодогенерации.

План

Ссылки

- Project Kepler, Compile-Time Metaprogramming for Scala
  https://github.com/xeno-by/kepler
- Живой журнал на русском, обновляется чаще всего
  http://xeno-by.livejournal.com/
- Блогспот на английском, ретрансляция из ЖЖ
  http://xeno-by.blogspot.com/
- Сборник познавательных текстов по макросам
  http://macros.xeno.by/
- Сайт языка программирования Немерле
  http://nemerle.org/

Благодарности

Пользуясь случаем, хочу выразить благодарность создателям языка программирования Немерле за прекрасный источник вдохновения, а также Владу Чистякову, главному разработчику современного Немерле, за невероятно полезные дискуссии на тему метапрограммирования времени компиляции.

Признаться честно, я просто восхищен Немерле. Этот язык программирования - живое доказательство того, что возможно и практично иметь систему интегрированного в язык метапрограммирования в статически типизированном языке с синтаксисом. Кроме того, Немерле является тестовым стендом для активных исследований в области макрологии, что делает его еще более интересным.

Наконец, большое спасибо всем, кто принимал участие в обсуждениях проекта “Кеплер” и своим взглядом со стороны помог сделать дизайн макросов лучше.

Обратная связь

Хотел бы еще раз остановиться на том, как важно ваше мнение.

Если после просмотра слайдов у вас возникли интересные идеи, если вы знаете, как макросы помогли бы (или помогают) вам в ежедневной работе, не сочтите за труд поделиться мыслями по адресу eugene.burmako@epfl.ch.

Макросы для Скалы все еще в процессе дизайна, поэтому удачная мысль может радикально поменять ход проекта и изменить мир к лучшему. Или же, более банально, но не менее приятно: сделать конкретно ваш сценарий программирования более продуктивным.

Вопросы и ответы

eugene.burmako@epfl.ch

Метаморфные макросы

```
macro Cons is Expr
  syntax expr1 : 11 "::" expr2 : 10;
{
  <[ @::(expr1, expr2) ]>
}
```

В отличие от обычных макросов, метаморфные макросы, работают на уровне парсера и грамматики языка программирования. Они могут добавлять новые синтактические конструкции или расширять уже существующие продукции грамматики.

Во второй версии языка команда Немерле планирует реализовать полностью весь синтаксис на макросах. Наряду с традиционными описанием грамматики DSL задания синтаксиса также будет включать информацию о типизации продукций. По сути, Немерле 2 будет метациркулярным метакомпилятором.