САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Математико-механический факультет

Кафедра системного программирования

Отладка макросов в языке программирования Scala

Дипломная работа студента 545 группы

Найданова Дмитрия Геннадьевича

Научный руководитель	.... /подпись/	д.ф.-м.н., проф. Терехов А.Н.
Рецензент	.... /подпись/	к.ф.-м.н. Булычев Д.Ю.
“Допустить к защите” заведующий кафедрой,	.... /подпись/	д.ф.-м.н., проф. Терехов А.Н.

Санкт-Петербург

2013

SAINT-PETERSBURG STATE UNIVERSITY

Mathematics and Mechanics Faculty

Chair of Software Engeneering

Macro debugging for Scala programming language

Graduation thesis

Naidanov Dmitrii

Scientific supervisor	.... /signature/	Full Professor Terekhov A.
Reviewer	.... /signature/	PhD, Associate professor Boulytchev D.
“Admitted for defence” Head of the Chair,	.... /signature/	Full Professor Terekhov A.

Saint Petersburg

2013

Содержание

1 Введение **5**

2 Постановка задачи **8**

3 Обзор существующих работ **9**

3.1 Отладка макросов в других языках программирования . . . 9

3.2 Многоуровневая отладка 10

4 Предлагаемый подход **11**

4.1 На стороне компилятора 11

4.1.1 Генерация синтетического кода 11

4.1.2 Генерация отладочной информации для байт-кода . . 11

4.1.3 Сохранение и передача синтетического кода отладчику 12

4.2 На стороне отладчика 13

5 Архитектура **14**

5.1 Подсистема компилятора 14

5.1.1 Создание синтетического исходного кода 14

5.1.2 Передача информации отладчику 15

5.1.3 Переписывание позиций в AST 15

5.2 Модуль внутри отладчика 16

5.2.1 Выбор отладчика 16

5.2.2 Обработка и хранение виртуальных файлов 18

5.2.3 Работа с API отладчика в IntelliJ IDEA 19

5.2.4 Некоторые дополнительные особенности 21

6 Переписывание позиций в абстрактных синтаксических деревьях **22**

6.1 Использования механизма установки позиций парсера на синтетическом коде. 22
6.2 Обход AST и переписывание позиций «вручную» 22

7 Заключение **24**

8 Список литературы **25**

1 Введение

Все современные языки программирования используют процедуры в том или ином виде. Процедура - именованная или иным образом идентифицированная часть компьютерной программы, содержащая описание определённого набора действий. Процедура может быть многократно вызвана из разных частей программы.

К сожалению, такая абстракция не всегда достаточно гибка, так как она ограничена синтаксисом и семантикой языка программирования. Например, в большинстве языков программирования невозможно определить ленивые логические операции в виде процедур, так как процедуры не могут влиять на то, в каком порядке выполняются вычисления. Другой пример — цикл for языка C, который поддерживает пролог с объявлением переменных, видимых в теле цикла.

Некоторые из перечисленных проблем могут быть решены при помощи макросов. Макросы — специальные процедуры, которые заменяют некоторые части программы по заданным правилам. Процесс применения макроса называется раскрытием макроса. Макросы делятся на лексические, которые работают с потоком лексем, и синтаксические, которые работают с абстрактными синтаксическими деревьями (далее — AST).

Макросы могут упростить решение некоторых сложных задач программирования. Например, с помощью макросов становится возможным использовать следующее:

1. Реификация (сохранение артефактов времени компиляции до времени исполнения, например исчезнувших в результате стирания параметров типов полиморфных классов в JVM).

2. Виртуализация (переопределение семантики языка для создания DSL).

3. Автоматическая генерация кода. Например, поставщики типов в F#.

Синтаксические макросы есть в языках Lisp, Ocaml, Nemerle и других. Реализацию синтаксических макросов содержит язык программирования Scala[4].

Scala - мультапарадигменный язык для JVM, разработанный в LAMP EPFL. Scala сочетает возможности функционального и объектно-ориентированного программирования. Первая версия появилась в 2003 году, версия 2.10, в которой впервые появились макросы, вышла 4 января 2013. Макросы в данный момент являются экспериментальной функциональностью [2].

```
//Macros.scala
import reflect.macros.Context
import scala.language.experimental.macros

object Macros {
  def helloDef(who: String) = macro Impl

  def helloImpl(c: Context)(who: c.Expr[String]) =
    c.universe.reify({
      println("Hello, " + who.splice)
    })
}

//MacroCall.scala
object MacroCall {
  def hello() = helloDef("World")
}
```

Пример макроса в языке Scala

Выше приведен пример простейшего макроса в Scala. Макрос состоит из двух частей - определения (`helloDef`) и реализации (`helloImpl`). Когда во время стадии проверки типов компилятор встретит вызов метода `helloDef`, он выполнит код реализации макроса с AST для литерала “`World`” в качестве аргумента. Кроме того, реализации будет передан спе-

циальный контекст, который содержит информацию, собранную компилятором. Результатом вызова макроса станет AST, которое будет подставлено на место вызова.

С отладкой макросов в языке Scala связаны некоторые проблемы. Если для кода, управляющего раскрытием макроса, еще можно использовать стандартные средства, предоставляемые платформой JVM [3], то, в силу следующих причин, для отладки полученных после раскрытия AST необходим некоторый специализированный механизм:

1. В результате раскрытия макроса получается AST. Но для отладки программы нужен исходный код, поэтому необходимо каким-то образом получить этот исходный код для AST (далее — синтетический исходный код).

2. Отладочная информация для синтетического исходного кода станет некорректной, т.к. будет соответствовать коду, написанному программистом (в котором макросы еще не раскрыты), и отладчик не сможет по ней правильно сопоставить синтетический код со сгенерированным байт-кодом.

2 Постановка задачи

1. Разработать и реализовать подсистему компилятора языка Scala для генерации необходимой отладочной информации и синтетического кода для AST, полученных в результате раскрытия макроса

2. Выбрать один из существующих отладчиков для языка Scala и реализовать в нем модуль для поддержки отладки макросов

3 Обзор существующих работ

3.1 Отладка макросов в других языках программирования

Реализация макросов в языке Scala во многих аспектах похожа [1] на реализацию в языке Nemerle [17].

Nemerle — гибридный язык высокого уровня со статической типизацией, сочетающий в себе возможности функционального и объектно-ориентированного программирования. Язык спроектирован для платформ .NET и Mono.

Nemerle позволяет создавать, анализировать и модифицировать код программы во время компиляции с помощью макросов. Макросы могут быть использованы либо в виде вызова метода, либо в виде новых конструкций языка. Большая часть конструкций в языке реализована с помощью макросов (if, for, foreach, while, using и т.д.).

Компилятор Nemerle генерирует достаточную информацию для отладки как кода, управляющего логикой раскрытия макроса [19], так и кода, получающегося в результате раскрытия [12].

Код, управляющий логикой раскрытия, может быть отлажен вместе с процессом компилятора или IDE.

Для кода, получающегося в результате раскрытия, можно указать компилятору явно сгенерировать исходный файл с синтетическим исходным кодом при помощи метода (`TypeBuilder.DefineWithSources()`). Полученный таким образом файл можно будет отлаживать как обычный исходный файл. С использованием этого способа связано несколько проблем:

1. Нельзя сгенерировать код для макросов уровня выражений.

2. Форматирование получаемого синтетического исходного кода не все-

гда корректно.

3. Метод `DefineWithSource()` создает строковое представление для AST сразу же после раскрытия макроса, до окончания стадии проверки типов. Код может измениться на следующих стадиях. Например, `def a = 1` превратится в `def a : int = 1`. В результате, код, который будет отлаживать программист, может существенно отличаться от того, который был скомпилирован.

3.2 Многоуровневая отладка

Помимо реализаций для конкретных языков, существуют работы, описывающие процесс обеспечения отладки кода, отличного от компилируемого кода, в общем.

Например, в [18] описывается подход для случая когда исходный код программы транслируется в промежуточный (т.н. объектный) код, который затем компилируется. При этом существует отладчик для объектного кода, задача состоит в том, чтобы проводить отладку в терминах исходного кода.

В спецификации JSR 45[9] описывается поддерживаемый JVM способ отладки языков кроме Java (например, JSP [11] — язык для написания шаблонов страницы) в терминах исходного кода. Спецификация позволяет задать несколько уровней исходного кода и промежуточных представлений, так называемых страт, и таблицу соответствий между ними. Например, для уже упомянутого JSP первой (основной) стратой будет шаблон, написанный на JSP, второй - код на Java, полученный после трансляции шаблона. Таблица соответствий записывается в атрибут SourceDebugExtension class-файла, который затем могут использовать JSR-45-совместимые отладчики.

4 Предлагаемый подход

4.1 На стороне компилятора

4.1.1 Генерация синтетического кода

Первый шаг используемого подхода — генерация синтетического кода. В компиляторе языка Scala существует стандартный механизм для генерации строкового представления AST. На этом шаге, используя этот механизм, для каждого раскрытого макросы получается синтетический исходный код. Далее заменой в тексте исходного файла всех вызовов макросов на соответствующий синтетический код получается синтетический исходный файл. Также сохраняется таблица соответствий номеров строк в старом и новом файлах.

4.1.2 Генерация отладочной информации для байт-кода

В JVM информация для отладки делится на три уровня:

1. Имя исходного файла
2. Таблица с отображением номеров строк в исходном файле на номера инструкций байт-кода
3. Имена локальных переменных

Имя исходного файла будет сгенерировано автоматически по имени синтетического файла. Имена локальных переменных будут сгенерированы по полученному после раскрытия AST, и они также будут сгенерированы корректно (т.е. так, чтобы они соответствовали тем, которые были в коде).

Таблицу с отображением необходимо исправить, так как компилятор создаст ее для исходного файла, причем все номера инструкций, полученных из раскрытого макроса, будут отображаться на строку с вызовом макроса.

Если этот вызов будет раскрыт в код, который займет несколько строчек, номера этих строчек в таблице "склеятся" в один номер инструкции байт-кода.

В качестве способа решения данной проблемы было выбрано переписывание позиций AST для всего файла, т.е. исправление позиции каждого узла в AST файла так, чтобы коду, полученному в результате раскрытия, соответствовали корректные позиции. Кроме этого, никаких модификаций не требуется, компилятор создаст правильную таблицу в фазе генерации байт-кода.

В качестве альтернативного подхода было рассмотрено использование JSR 45. На практике, этот способ не дал никаких преимуществ, так как из-за проблемы "склеивания" номеров инструкций все равно потребовалось исправлять позиции в AST.

С переписыванием позиций AST связана проблема искажения отладочной информации относительно "обычного" исходного файла. Отладочная таблица используется также для сопоставления трассировки стека с номерами строк в исходном коде при возникновении исключений, профилировании и т.п., это приводит к тому, что байт-код, полученный из AST с переписанными позициями, нельзя поставлять пользователям. Поэтому в настройки компилятора был добавлен флаг для влючения или отключения отладки макросов.

4.1.3 Сохранение и передача синтетического кода отладчику

После выполнения двух предыдущих шагов появляются два артефакта: синтетический исходный файл и таблица соответствий номеров символов. Оба артефакта сохраняются для дальнейшего использования отладчиком.

4.2 На стороне отладчика

Отладчик должен загрузить синтетический исходный код, сгенерированный компилятором, обработать его для получения виртуального файла в своем внутреннем представлении и использовать этот файл для своей функциональности (обработка точек прерывания, трассировка (stepping) и т.п.)

5 Архитектура

5.1 Подсистема компилятора

Подсистема компилятора реализована в виде дополнительной фазы. Такой способ выбран из соображений обратной совместимости: компилятор, кроме “обычного” режима, может использоваться как компилятор представления (presentation compiler) [16]. Этот режим предназначен для использования внутри интегрированной среды разработки, в нем можно получить доступ к промежуточной информации для реализации такой функциональности как подсветка синтаксиса, автодополнение и т.п. Последняя фаза, которую использует компилятор представления — фаза проверки типов, фаза генерации отладочной информации для макросов вставлена сразу после нее.

5.1.1 Создание синтетического исходного кода

Сразу после фазы проверки типов происходит генерация синтетического кода.

Для того, чтобы сохранить всю необходимую информацию до стадии генерации отладочной информации, во время фазы проверки типов для каждого успешного раскрытого макроса в глобальный промежуточный буфер сохраняются позиция исходного вызова макроса и ссылка на полученное в результате раскрытия дерево.

В самом начале новой фазы, если в предыдущей фазе внутри текущего файла был раскрыт хоть один макрос, из буфера извлекаются все деревья и позиции, относящиеся к этому файлу. Далее создается виртуальный файл, в который копируется код исходного файла с вставленным строковым представлением соответствующих AST на места вызовов макросов. Виртуальный файл отмечается как исходный файл для текущей единицы

компиляции компиляции (compilation unit).

5.1.2 Передача информации отладчику

После создания виртуального файла, текст этого файла отправляется в стандартный поток вывода. Перед каждой строкой текста вставляется специальный префикс, который можно указать в настройках компилятора. По этому префиксу отладчик отличит текст синтетического файла. После в стандартный поток выводятся список соответствий отступов в старом и новом исходных файлах в виде (позиция вызова макроса, длина строкового представления получившегося дерева).

5.1.3 Переписывание позиций в AST

После того, как виртуальный файл был сгенерирован, то следующим шагом производится переписывание позиций AST всей единицы компиляции.

Это сделано для того, чтобы компилятор мог в фазе генерации байт-кода по новым позициям сгенерировать правильную информацию для отладки. Без этого шага потребовалась бы модификация расчета номеров строк в фазе генерации промежуточного представления по AST.

Плюсы данного подхода:

1. Прозрачность для остальных инструментов.

2. Простота реализации (в том смысле, что изменения требуются только в одной части компилятора).

Минусы:

1. Необходимость отдельной компиляции для включения отладки макросов. Т.е. если проект был скомпилирован без включенной настройки для генерации отладочной информации для макросов, а потом

все же понадобилось отлаживать макросы, то весь проект придется скомпилировать заново.

2. Невозможно отлаживать и поставлять один и тот же байт-код из-за проблем с соответствием с “обычным” исходным файлом.

Альернативный подход — использование JSR 45. В теории, он позволил бы избавиться от этих недостатков, т.к. для виртуального исходного файла можно создать дополнительную страту, а в отладчике выбирать, страту для какого файла использовать. На практике, необходимость отдельной компиляции не является большим недостатком, т.к., во-первых, как правило проект всегда пересобирается перед поставкой, во-вторых, отдельная компиляция всегда нужна для генерации синтетического кода. Кроме того, использование JSR 45 имеет ряд недостатков:

1. Использование JSR 45 ограничивает выбор отладчиков.
2. Реализация будет технически более сложной.

В силу этих причин, было решено отказаться от использования JSR 45.

5.2 Модуль внутри отладчика

5.2.1 Выбор отладчика

Большинство отладчиков для языка Scala — встроенные отладчики в интегрированных средах разработки (далее — IDE). Было рассмотрено две такие IDE — IntelliJ IDEA и Scala IDE for Eclipse.

IntelliJ IDEA

IntelliJ IDEA — интегрированная среда разработки программного обеспечения, которая поддерживает большое количество языков, в частности

Java и Scala, разработанная компанией JetBrains. Есть проприетарная и открытая версии, официальный плагин для поддержки языка Scala доступен в обеих [5].

IDEA предоставляет API для написания собственного отладчика для любых компилирующихся в JVM байт-код языков, такой отладчик реализован в плагине для языка Scala. Также этот плагин предоставляет[15]:

- подсветку синтаксиса кода;
- автодополнение;
- поиск ссылок на символ;
- поиск декларации символа;
- подсветку ошибок в коде;
- различные рефакторинги;
- форматирование кода;
- интеграцию с тестовыми фреймворками JUnit, TestNG, ScalaTest и Specs;
- и другую функциональность.

Scala Plugin for IntelliJ IDEA содержит свою реализацию части фронтенда компилятора и не использует компилятор представления.

Scala Plugin использует формат внутреннего представления структуры кода IntelliJ IDEA, API для работы с файловой системой и другие интерфейсы, т.е. вся документация, описывающая эти объекты в IntelliJ IDEA применима также и к Scala-плагину.

Scala IDE for Eclipse

Eclipse — интегрированная среда разработки модульных кроссплатформенных приложений. Развивается и поддерживается Eclipse Foundation.

Наиболее известные приложения на основе Eclipse Platform — различные IDE для разработки ПО. Среди них есть IDE для Scala. Также, как и Scala Plugin for IntelliJ IDEA, Scala IDE содержит подсветку кода и ошибок, поиск деклараций и т.п.

Scala IDE использует компилятор представления.

Scala IDE for Eclipse использует формат компилятора для представления структуры кода и свои собственные форматы для некоторой другой функциональности (например, форматтер и отладчик) [14]. Документации по этим системам меньше, чем у IntelliJ IDEA.

Так как формат представления структуры кода в Scala Plugin for IntelliJ IDEA соответствует стандартному формату, используемому повсеместно в IntelliJ IDEA и плагинах к ней, то для него доступно большое количество вспомогательных классов, стандартных методов и т.п., что делает более удобной работу с программой. Также весь этот стандартный формат и большая часть API IntelliJ IDEA, в отличие от Scala IDE, достаточно хорошо документированы. В силу этих причин была выбрана IntelliJ IDEA.

5.2.2 Обработка и хранение виртуальных файлов

После компиляции кода с включенной отладкой макросов отладчик извлекает из стандартного потока вывода компилятора набор пар артефактов (синтетический исходный файл с указанием, к какому файлу он относится; соответствие отступов в исходном файле и в синтетическом файле).

Работа с файлами в IDEA, в основном, происходит на двух уровнях — Virtual File System (VFS) [8] и Program Structure Interface (PSI) [6].

VFS [8] инкапсулирует большую часть работы с файлами, она предоставляет:

- Средства для работы с файлами, которые позволяют одинаково работать с файлами, вне зависимости от их места расположения.
- Возможность отслеживания времени модификации и историю файла.
- Возможность связать с файлом некоторую информацию посредством атрибутов. Сохраненная в атрибутах информация будет персистентна [7], т.е. будет сохраняться между запусками среды разработки.

Внутренний API IntelliJ IDEA позволяет создавать виртуальные файлы, т.е. файлы, которые находятся в памяти и которые доступны только из самой IDE. Такой виртуальный файл создается для каждой пары артефактов, полученной от компилятора. Также для каждого файла, содержащего вызов хотя бы одного макроса, в атрибутах этого файла сохраняется соответствующий ему виртуальный файл.

PSI [6] — представление внутренней структуры исходного кода в терминах IntelliJ IDEA. Оно может использоваться для статического анализа, рефакторингов, вывода типов и т.п.. Для представления файла на этом уровне используется класс `PsiFile`.

5.2.3 Работа с API отладчика в IntelliJ IDEA

IntelliJ IDEA работает с Java Debugger Interface, самым верхним уровнем Java Platform Debugger Architecture (JPDA) [10]. Для абстракции от JDPA в интегрированной среде предлагается реализовать интерфейс `PositionManager`, который в том числе содержит методы для отображения отладочной информации (имени исходного файла и номера строки исходного кода), полученной от JPDA, на код внутри IDE.

Для языка Scala реализован `ScalaPositionManager`. Он содержит реализацию всех методов `PositionManager`, а также некоторые утилитные методы. Среди них есть метод, который находит `PsiFile`, соответствующий позиции в байт-коде. Он был дописан так, чтобы при нахождении для

позиции некоторого файла, включенной отладке макросов и наличии в атрибутах виртуального файла он возвращал найденный виртуальный файл. Таким образом в нужных случаях для отладчика языка Scala "обычный" исходный файл подменяется на виртуальный, а т.к. виртуальный файл с точки зрения API среды ничем не отличается от любого другого файла, никаких дополнительных модификаций внутри отладчика не требуется.

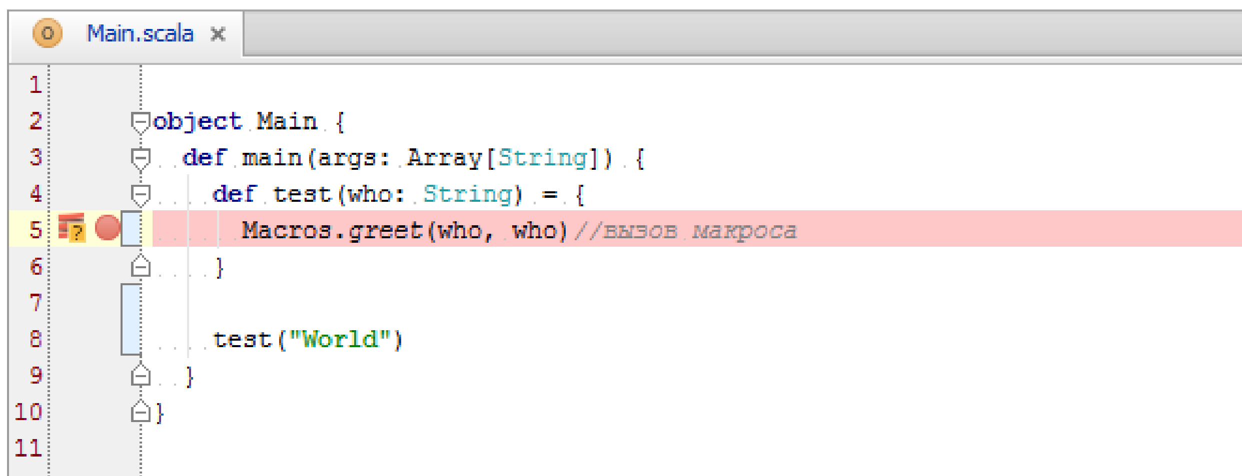

Рис. 1. Окно редактора

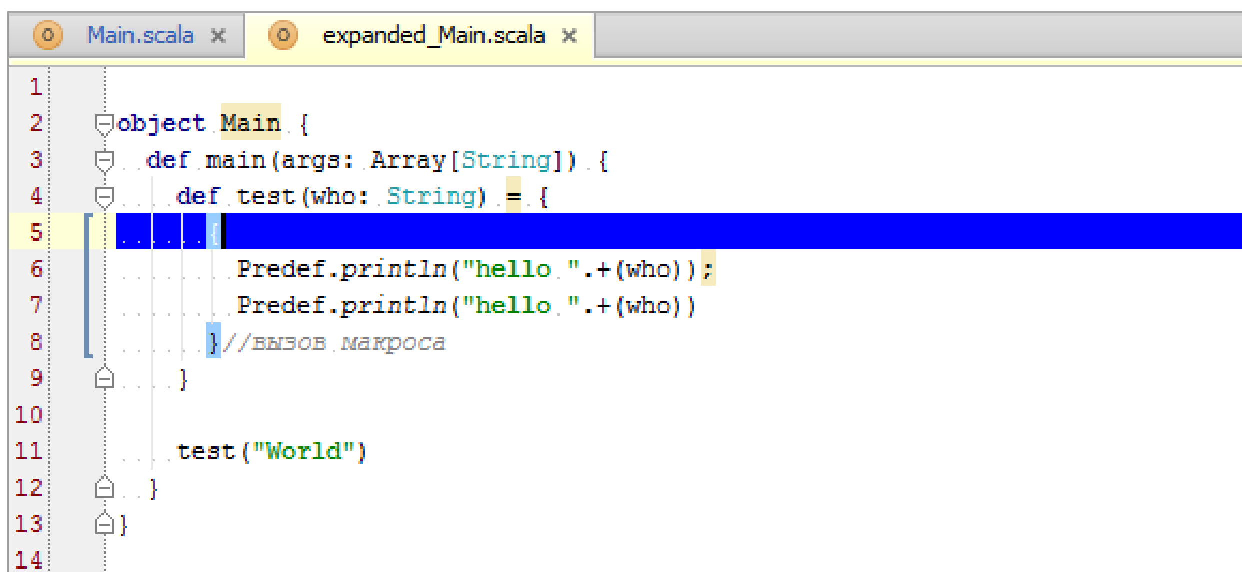

Рис. 2. Окно редактора после срабатывания точки прерывания

5.2.4 Некоторые дополнительные особенности

- При срабатывании точки прерывания, которая относится к виртуальному файлу, происхоит навигация в этот файл, с открытием новой вкладки редактора, если нужно.
- На боковой панели редактора после компиляции напротив каждого вызова макроса появляется маркер, при наведении курсора на который показывается код, полученный при раскрытии макроса.
- При нажатии на маркер происходит навигация в соответствующее место виртуального файла.

6 Переписывание позиций в абстрактных синтаксических деревьях

Были рассмотрены два подхода:

6.1 Использования механизма установки позиций парсера на синтетическом коде.

Чтобы получить правильные позиции для AST (1), можно на его текстовом представлении запустить парсер, получить еще одно AST (2), затем параллельно обойти оба дерева и скопировать позиции с (1) в (2). Такой подход даст возможность расставить позиции абсолютно правильно (в том смысле, что такая расстановка не отличалось бы от расстановки, полученной при “обычной” компиляции синтетического кода), также он относительно прост в реализации. Но у такого подхода есть ряд недостатков:

1. Придется создать еще одно AST для всего файла.
2. На практике выяснилось, что в некоторых случаях дерево (2) не совпадает с исходным деревом, т.е. нельзя для каждого узла дерева (1) найти соответствующий узел дерева (2).

Из-за №2. было решено использовать второй подход.

6.2 Обход AST и переписывание позиций «вручную»

Второй способ — обойти все дерево и пересчитать позицию для каждого узла.

Пусть N — некоторый узел AST, тогда за $N.pos$ обозначим его позиции. Позиции узла AST содержат начало и конец соответствующего узлу

текста в исходном файле и т.н. точку (специфическую для каждого типа узла позицию в тексте, например, позицию скобки в вызове метода; точка используется, в частности, при отображении позиции при сообщении об ошибке компиляции), обозначим их за $N.pos.start$, $N.pos.end$ и $N.pos.point$ соответственно. Разницу $N.pos.end - N.pos.start$ будем называть длиной позиции. Пусть M - некоторое определение макроса, тогда $M.call$ — его вызов, $M.call.expLength$ — длина текстового представления AST, получившегося в результате его раскрытия.

Правила переписывания позиции для некоторого узла N выглядят так:

1. Если концу узла $N.pos.end$ не предшествует ни один вызов макроса, то позицию N менять не надо.

2. Если $N.pos.start, N.pos.point$ или $N.pos.end$ находятся после вызова хотя бы одного макроса, то их нужно сдвинуть на сумму длин текстовых представлений AST, получившихся в результате раскрытия всех макросов, вызовы которых находились до них, т.е. $\forall i \in \{start, point, end\}\ \forall M.call_j : i > M.call_j.pos, i := i + \sum_j (M.call.expLength - M.call_j.length)$.

3. У узла N синтаксического дерева, полученного от раскрытия макроса, и всех дочерних узлов N_i позиции будут равны позиции вызова макроса, из которого это дерево получилось. Тогда каждому узлу N, N_i нужно сопоставить позицию, такую что: длина позиции равна длине текстового представления узла, начало равно началу исходной позиции + сумма длин текстовых представлений всех предшествующих дочерних узлов + сумм длин текстовых представлений всех AST, получившихся в результате раскрытия макросов, вызовы которых предшествовали N. “Точка” должна быть проставлена также, как и в парсере, т.е. по таким же правилам, по каким присваивается *point* позиции во время синтаксического анализа исходного файла.

7 Заключение

В рамках данной дипломной работы были выполнены следующие задачи:

1. Разработан подход для отладки кода, полученного в результате раскрытия макроса в компиляторе языка Scala

2. Реализованы механизмы для отладки на стороне компилятора и на стороне встроенного отладчика в IntelliJ IDEA

В дальнейшем планируется исправить обнаруженные ошибки и оформить оформить pull request в репозиторий компилятора на github.com в соответствии с официальными рекомендациями[13].

Список литературы

[1] Burmako E. Scala Macros — project Kepler. — URL: scalamacros.org.

[2] Metaprogramming with Macros, Research Proposal : Rep. / LAMP EPFL ; Executor: E. Burmako : 2012.

[3] Burmako E. Scala Macros Documentation. — 2012. — URL: http://docs.scala-lang.org/overviews/macros/overview.html.

[4] Burmako E. Scala Macros: Let Our Powers Combine. — 2013.

[5] Getting Started with IntelliJ IDEA Scala Plugin. — URL: http://confluence.jetbrains.com/display/SCA/Getting+Started+with+IntelliJ+IDEA+Scala+Plugin.

[6] IntelliJ IDEA Architectural Overview. — URL: http://confluence.jetbrains.com/display/IDEADEV/IntelliJ+IDEA+Architectural+Overview.

[7] IntelliJ IDEA Persisting State of Components. — URL: http://confluence.jetbrains.com/display/IDEADEV/Persisting+State+of+Components.

[8] IntelliJ IDEA Virtual File System. — URL: http://confluence.jetbrains.com/display/IDEADEV/IntelliJ+IDEA+Virtual+File+System.

[9] JSR 45: Debugging Support for Other Languages. — 2003. — URL: http://jcp.org/en/jsr/detail?id=45.

[10] Java Platform Debugger Architecture. — URL: http://docs.oracle.com/javase/7/docs/technotes/guides/jpda/index.html.

[11] Java Server Pages. — URL: http://www.oracle.com/technetwork/java/javaee/jsp/index.html.

[12] Nemerle — интеграция с VS 2008. — URL: http://rsdn.ru/article/?857.

[13] Scala — Pull Request Policy. — URL: https://github.com/scala/scala/wiki/Pull-Request-Policy.

[14] Scala IDE for Eclipse Architecture. — URL: http://scala-ide.org/docs/dev/architecture/architecture.html.

[15] Scala Plugin features. — URL: http://confluence.jetbrains.com/display/SCA/Scala+Plugin+0.2+features.

[16] The Scala Presentation Compiler. — URL: http://scala-ide.org/docs/dev/architecture/presentation-compiler.html.

[17] Skalski K. Syntax-extending and type-reflecting macros in an object-oriented language. — 2005.

[18] Карташев М. С. Двухуровневая система отладки // Системное программирование. Вып. 1: Сб. статей под ред. А.Н.Терехова, Д.Ю.Булычева. — 2004. — Т. 1. — С. 181.

[19] Чистяков В. Макросы Nemerle – расширенный курс // RSDN Magazine. — 2007. — Т. 1. — С. 76.