Министерство образования и науки Российской Федерации
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ»

МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ

«Compiler-agnostic модель раскрытия макроопределений с динамическим выводом типов языка программирования *Scala*»

Автор: Муцянко Михаил Сергеевич _____

Направление подготовки (специальность): 01.04.02 Прикладная математика и информатика

Квалификация: Магистр

Руководитель: Буздалов М.В., канд. техн. наук _____

К защите допустить

Зав. кафедрой Васильев В.Н., докт. техн. наук, проф. _____

«____» _____ 20____ г.

Санкт-Петербург, 2016 г.

Студент Муцянко М.С. **Группа** М4238 **Кафедра** компьютерных технологий **Факультет** информационных технологий и программирования

Направленность (профиль), специализация Технологии проектирования и разработки программного обеспечения

Квалификационная работа выполнена с оценкой _____

Дата защиты «____» _____ 20_____ г.

Секретарь ГЭК _____

Листов хранения _____

Демонстрационных материалов/Чертежей хранения _____

Министерство образования и науки Российской Федерации
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ»

УТВЕРЖДАЮ
Зав. каф. компьютерных технологий
докт. техн. наук, проф.
_____ Васильев В.Н.
«____» _____ 20_____ г.

ЗАДАНИЕ
НА МАГИСТЕРСКУЮ ДИССЕРТАЦИЮ

Студент Муцянко М.С. **Группа** М4238 **Кафедра** компьютерных технологий **Факультет** информационных технологий и программирования **Руководитель** Буздалов Максим Викторович, канд. техн. наук, научный сотрудник Университета ИТМО

1 Наименование темы: Compiler-agnostic модель раскрытия макроопределений с динамическим выводом типов языка программирования *Scala*

Направление подготовки (специальность): 01.04.02 Прикладная математика и информатика

Направленность (профиль): Технологии проектирования и разработки программного обеспечения

Квалификация: Магистр

2 Срок сдачи студентом законченной работы: «____» _____ 20_____ г.

3 Техническое задание и исходные данные к работе.

Требуется разработать такой метод, который бы позволил реализовать compiler-agnostic модель раскрытия макроопределений с динамическим выводом типов языка программирования *Scala* на платформе IntelliJ Idea.

4 Содержание магистерской диссертации (перечень подлежащих разработке вопросов)

Разработанный метод и его образец реализации на платформа IntelliJ Idea должен демонстрировать соответствие стандартам как целевой платформы IntelliJ Idea так и платформе метапрограммирования *scala.meta*, а также быть удобным в использовании и показывать производительность, соответствующую возможности использования в интерактивном режиме.

5 Перечень графического материала (с указанием обязательного материала)

Не предусмотрено

6 Исходные материалы и пособия

а) «*Scala* Language Specification» http://www.scala-lang.org/files/archive/spec/2.11/;

б) *scala.meta* specs http://scalameta.org/;

в) Исходные коды обоих целевых платформ.

7 Календарный план

№№ пп.	Наименование этапов магистерской диссертации	Срок выполнения этапов работы	Отметка о выполнении, подпись руков.
1	Ознакомление с кодовой базой IntelliJ Idea	10.2015	
2	Ознакомление с кодовой базой *scala.meta*	10.2015	
3	Построение общих концепций и базовых принципов работы платформы	12.2015	
4	Реализация первоначальной версии платформы	02.2016	
5	Написание пояснительной записки	03.2016	
6	Доработка платформы и пояснительной записки	05.2016	

8 Дата выдачи задания: «____» _____ 20_____ г.

Руководитель _____

Задание принял к исполнению _____ «____» _____ 20_____ г.

Министерство образования и науки Российской Федерации
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ»

АННОТАЦИЯ
МАГИСТЕРСКОЙ ДИССЕРТАЦИИ

Студент: Муцянко Михаил Сергеевич
Наименование темы работы: Compiler-agnostic модель раскрытия макроопределений с динамическим выводом типов языка программирования *Scala*
Наименование организации, где выполнена работа: Университет ИТМО

ХАРАКТЕРИСТИКА МАГИСТЕРСКОЙ ДИССЕРТАЦИИ

1 Цель исследования: Построение такого метода, который позволил бы интегрировать платформу метапрограммирования *scala.meta* в среде разработки IntelliJ Idea
2 Задачи, решаемые в работе:

а) выполнять преобразования синтаксических деревьев между платформами;
б) предоставлять семантическую информацию о преобразовываемых деревьях;
в) реализовывать семантические методы *scala.meta*.

3 Число источников, использованных при составлении обзора: _____
4 Полное число источников, использованных в работе: 10
5 В том числе источников по годам

Отечественных			Иностранных		
Последние 5 лет	От 5 до 10 лет	Более 10 лет	Последние 5 лет	От 5 до 10 лет	Более 10 лет

6 Использование информационных ресурсов Internet: _____
7 Использование современных пакетов компьютерных программ и технологий: Был использован язык программирования *Scala*, интегрированная среда разработки IntelliJ Idea, системы автоматического тестирования JUnit и scalatest а также система непрерывной интеграции TeamCity.
8 Краткая характеристика полученных результатов: В результате была получена система, выполняющая поставленные требования и интегрируемая в целевую платформу.
9 Гранты, полученные при выполнении работы: Грантов или других форм государственной поддержи и субсидирования в процессе работы не предусматривалось.
10 Наличие публикаций и выступлений на конференциях по теме работы: Первая публичная демонстрация промежуточных результатов, полученных в рамках данной работы, была проведена на конференции Scala Days 2016 Berlin.

Выпусник: Муцянко М.С. _____

Руководитель: Буздалов М.В. _____

«____» _____ 20_____ г.

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Большинство современных языков программирования являются мультипарадигменными и предоставляют программисту возможность выбирать принципиально разные подходы к реализации поставленных задач. Одним из таких подходов является метапрограммирование, позволяющий, к примеру, создавать программы, результатом работы которых являются другие программы, что позволяет, в перспективе, прикладывать меньшие усилия при разработке ПО, нежели в случае, когда весь исходный код необходимо писать вручную.

Одним из языков, поддерживающих данную парадигму, является *Scala* [1]. Данный язык предоставляет более широкую поддержку метапрограммирования чем большинство прочих промышленных языков благодаря механизму макросов, позволяющих выполнять обширные трансформации абстрактного синтаксического дерева программы на этапе компиляции. К примеру, в отличие от макросов языка *C/C++* [2], которые обрабатываются отдельной программой-препроцессором, и, как следствие, не имеют такого представления кода программы, которое имел бы компилятор в процессе работы, макросы языка *Scala* являются подмножеством самого языка и выполняются в контексте компилятора в качестве совмещённой с тайпчекером фазы компиляции и имеют такой же доступ к информации о программе, как и сам тайпчекер.

Однако изначальная реализация макросистемы языка *Scala* обладает рядом существенных недостатков. Таких как, например, значительное количество избыточного кода, необходимого для написания метапрограмм, пререусложненность и недостаточная ясность API для работы с абстрактными синтаксическими деревьями и манипуляции семантической информацией привязанной к ним. Помимо вышеобозначенных проблем также существует проблема, связанная с необходимостью раздельной компиляции тел макросов и исходных файлов, в которых они используются, что является следствием специфичных архитектурных решений, применяемых в текущей системе метапрограммирования языка *Scala*.

В целях разрешения данных проблем в Федеральной политехнической школе Лозанны была создана новая система метапрограммирования под названием *scala.meta*[3], которая абстрагируется от решений применённых в текущей реализации, а также от самого компилятора языка программирования *Scala*, вводя новый структуры данных для синтаксических деревьев, уменьшая количество избыточных абстракций предоставляющих синтаксическую и семантическую информацию, предоставляя возможность разработчикам различных платформ предоставлять собственные реализации семантических методов, таких как вывод типов, проверка соответствия, разрешение ссылок и другие.

Целью данной работы является создание метода, позволяющего выполнять преобразования деревьев, предоставлять необходимую семантическую информацию в контексте работы метапрограмм, использующих новую систему метапрограммирования языка *Scala* для платформы IntelliJ Idea.

ГЛАВА 1. ОБЗОР ПРЕДМЕТНОЙ ОБЛАСТИ

1.1. Макросы в языках программирования

Во многих языках программирования являются средствами метапрограммирования на этапе компиляции. Макросы можно условно подразделить на два вида:

— основанные на лексической токенизации;
— основанные на обработке синтаксических деревьев[4].

К первой категории относятся макросы в таких языках как *C/C++*, язык ассемблера, *MacroML*. Системы макросов, обозначенные выше, оперируют лексическими токенами и не могут надёжно сохранять структуру обрабатываемой программы.

Синтаксические макросы, в свою очередь, оперируют абстрактными синтаксическими деревьями и полностью сохраняют лексическую структуру исходной программы. Наиболее часто используемые реализации синтаксических макросов можно найти в *Lisp*-подобных языках, таких как *Common Lisp*, *Clojure*, *Scheme*, *ISLISP* и *Racket*[5]. Синтаксические макросы позволяют преобразовывать структуру программы, используя все возможности языка, в котором они реализованы. В последствии, данный подход распространился и на другие языки, такие как *Prolog*, *Dylan*, *Nemerle*, *Rust* и *Scala*.

1.2. Макросы языка *Scala*

Макросы в языке программирования *Scala*[6], так же, как и во многих других языках являются средствами, выполняющимися на этапе компиляции. Важной особенностью макросов *Scala* является форма входных и выходных данных представляющая собой типизированные синтаксические деревья. Макросы *Scala*, в общем случае, являются методами, вызовы которых раскрываются на этапе компиляции[7]. В данном контексте, под раскрытием подразумевается преобразование фрагмента кода полученного из вызова метода и его аргументов. Для упрощения написания макросов и экономии времени при составлении синтаксических деревьев применяются квазицитаты[8].

Листинг 1 - Пример assert макроса *Scala*

```
def assert(cond: Boolean, msg: String) = macro assertImpl
def assertImpl(c: Context) = {
  import c.universe._
  val q"assert($cond, $msg)" = c.macroApplication
  q"if (!$cond) raise($msg)"
}
```

Чтобы конкретизировать важные различия межу типами макросов, вводятся специальные обозначения для них.

1.2.1. BlackBox макросы

Макросы, которые имеют поведение, аналогичное обычным методам и имеют реальный тип возвращаемого значения эквивалентный типу, указанному в сигнатуре соответствующего им метода, называются BlackBox макросами

1.2.2. WhiteBox макросы

В некоторых случаях определение макроса может выходить за рамки определения обыкновенного метода. К примеру, макрос при раскрытии может породить терм, тип которого сужает тип возвращаемого значения, объявленного в самом макросе. Рассмотрим простой пример:

Листинг 2 - Пример whitebox макроса

```
def foo: Any = macro impl
def impl(c: Context): c.Expr[Unit] = q"2"
val x = foo // x:Int
```

Как видно из примера выше, тип выражения, полученного в результате раскрытия макроса, не соответствует типу, указанному в методе.

1.3. *scala.meta*

Как следствие многочисленных проблем и неудобств, возникающих при использовании оригинальной макросистемы языка *Scala*, была создана инициатива, направленная на разрешение данных проблем. Ее целью стало создание системы, позволяющей как упростить API самих

синтаксических деревьев, так и упразднить избыточность многих структур данных и их иерархий, сделать систему метапрограммирования достаточно гибкой, чтобы она могла позволять интегрироваться не только в компиляторе самого языка *Scala*, но также дать потенциальную возможность использования системы и в других версиях *Scala*, например Dotty, а также получить возможность интеграции в такие программные продукты как, например, интегрированные среды разработки, в первую очередь в IntelliJ Idea.

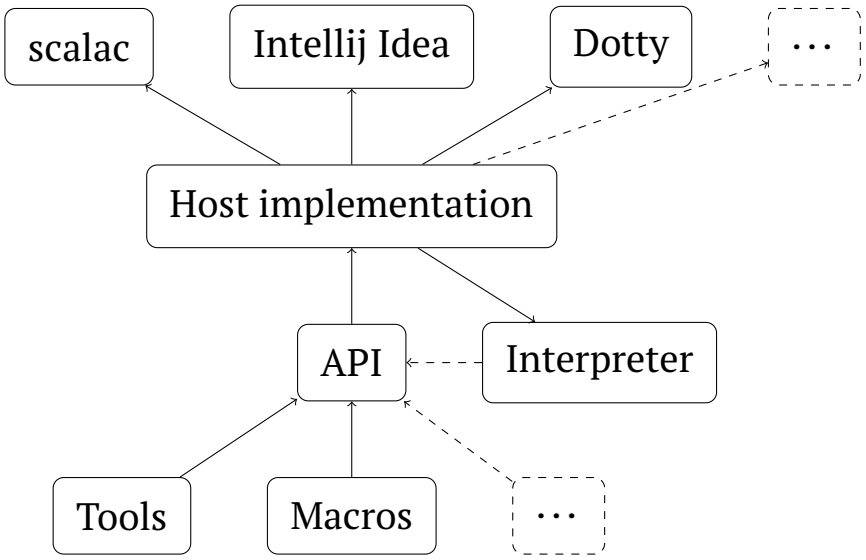


Рисунок 1 - *scala.meta*

Система метапрограммирования *scala.meta* предоставляет два основных интерфейса — внешний, используемый пользователями при написании метапрограмм и другой функциональности, а также внутренний, реализующий механизмы предоставления семантической информации о деревьях. В данной роли может выступать как компилятор соответствующего языка, так, например, и среда разработки, имеющая функционал автономного разрешения ссылок и вывода типов.

С пользовательской точки зрения новое API дает значительное число преимуществ по сравнению с существующей системой метапрограммирования, основанной на scala.reflect:

— сокращение количества утилитарного кода, необходимого в таких случаях, как, например, написание макросов;
— упразднение параллельных иерархий таких как отдельные иерархии для деревьев, символов, имён, и так далее;

— парсер без утери позиций и другой значимой синтаксической информации;
— отсутствие необходимости раздельной компиляции макросов — встраивание тела прямо в месте объявления;
— упрощение семантических операций — большинство задач реализуются двумя-тремя методами контекста.

В то время как практически вся функциональность синтаксического API может быть реализована внутри общей библиотеки *scala.meta*, семантическое API потребовало бы внедрения в общую библиотеку как полной функциональности по выводу и сравнению типов — тайпчекера, так и функциональности разрешения имен, что, в свою очередь, не только чрезвычайно увеличивает объем необходимой работы для их реализации, а также сигнализирует о нерациональном расходе ресурсов вследствие практически полного дублирования функциональности компилятора, но и заставляет постоянно поддерживать соответствие поведения этих систем между системой метапрограммирования и целевыми платформами.

1.4. Постановка задачи

Выбранный подход к реализации новой системы метапрограммирования для языка *Scala* вынуждает разработчиков целевых платформ, каждая из которых использует не только собственные наборы структур данных и алгоритмов для их обработки, но и собственные обобщенные подходы к решению множества задач, ключевых для любого инструмента работающего с языками программирования, будь то интегрированная среда разработки или компилятор того или иного языка, осуществлять достаточно нетривиальную трансляцию соответствующих структур данных, содержащих полную синтаксическую и семантическую информацию о исходном коде, который может как непосредственно обрабатываться и трансформироваться соответствующей метапрограммой, так и неявно использоваться ей в процессе работы. Помимо самих преобразований синтаксических деревьев, целевая платформа должна также динамически предоставлять разнообразную семантическую информацию, в добавок с статически разрешенной на этапе преобразования синтаксических деревьев.

Таким образом, в контексте данной работы предлагается создать такой комплекс подходов, который бы позволил решить все вышеозначенные задачи а также предоставить образец реализации данных методов в качестве одной из подсистем *Scala* плагина платформы IntelliJ Idea.

ГЛАВА 2. ПРЕДЛАГАЕМЫЙ МЕТОД

В данном разделе описан предлагаемый метод для решения задачи, поставленной в текущей работе.

2.1. Преобразование деревьев

Ключевым моментом интеграции системы метапрограммирования *scala.meta* в экосистему анализа исходного кода IntelliJ Idea, и в ее реализацию для языка *Scala* в частности, является преобразование синтаксических деревьев между двумя независимыми форматами [9]. При выполнении метапрограмм, написанных с использованием системы метапрограммирования *scala.meta* происходит выполнение набора преобразований и прочих операций над синтаксическими деревьями *scala.meta*. Причем при запуске таких метапрограмм внутри среды разработки IntelliJ Idea синтаксические деревья, поступающие на вход метапрограммам предоставляются самой средой, и, как следствие, требуют преобразования в соответствующий формат. Синтаксические деревья используемые в платформе IntelliJ Idea обладают рядом характерных особенностей:

— ленивое разрешение ссылок;
— ленивый вывод типов;
— неявное взаимодействие с деревьями других языков;
— возможность агрегировать деревья из декомпилированного байткода и других источников.

2.1.1. Типы синтаксических деревьев *scala.meta*

Прежде чем приступить к описанию деталей реализации конвертера, необходимо, для начала, прояснить структуру конвертируемых данных. В дальнейшем, запись синтаксических деревьев *scala.meta* будет вестись в форме указанной ниже. Рассмотрим основные типы деревьев, наиболее часто встречающихся при конвертации:

Tree

Корень иерархии классов деревьев *scala.meta*. Предоставляет возможности, общие для наследников, такие как, например релизация интерфейса `Serializable`.

Symbol

Символы предоставляют частичную семантическую информацию о деревьях, к которым они присоединены. Символ уникален для каждого уникального объявления , как, например, переменной, поля класса или метода.

Select (*qualifier*: Tree, *name*: Name)

Дерево выбора элемента с именем *name* из дерева *qualifier*. Например выбора поля или метода класса.

Term.Apply (*fun*: Term, *args*: Seq[Term.Arg])

Применение функции с аргументами *args*. Зачастую в качестве *fun* выступает дерево `Select`, задающее применяемый метод.

Term.New (*templ*: Template)

Представляет операцию `new`, инстанциирующую определение *templ*. Стоит отметить, что данное дерево не эквивалентно вызову конструктора.

Term.While (*expr*: Term, *body*: Term)

Цикл с предусловием.

If (*cond*: Term, *thenp*: Term, *elsep*: Term)

Оператор условного перехода. Конструкции `else if` преобразуются в последовательность вложенных конструкций `if`.

Term.Match (*scrut*: Term, *cases*: Seq[Case])

Представляет операцию сопоставления значения *scrut* с образцми, указанными в *cases*.

Try (*expr*: Term, *catchp*: Seq[Case], *finallyp*: Option[Term])

Представляет блок выражений `try ... catch ... finally`.

Throw (*expr*: Term)

Выбрасывает значение *expr* в качестве исключения.

Defn.Def (*mods*: Seq[Mod], *name*: Term.Name, *tparams*: Seq[Term.Param], *paramss*: Seq[Seq[Term.Param]], *decltpe*: Option[Type], *body*: Term)

Содержит объявление функции с телом *body* и именем *name*.

Decl.Val(*mods*: Seq[Mod], *pats*: Seq[Pat.Var.Term], *decltpe*: Type)

Дерево объявления последовательности паттернов *pat*.

Decl.Type(*mods*: Seq[Mod], *name*: Type.Name, *tparams*: Seq[Type.Param], *body*: Type)

Дерево объявления и инициализации алиаса типов.

2.1.2. Типы синтаксических деревьев IntelliJ Idea

В отличие от синтаксических деревьев *scala.meta*, на синтаксические деревья платформы IntelliJ Idea[10] не накладывается жестких ограничений по неизменяемости и алгебраичности структур данных, через которые данные деревья реализованы.

Иерархия классов синтаксических деревьев платформы IntelliJ Idea построена так, чтобы быть легко расширяемой при реализации требуемой функциональности анализа кода для множества языков программирования. В частности, основой для всех синтаксических деревьев является интерфейс PsiElement.

Листинг 3 - Интерфейс PsiElement

```
public interface PsiElement extends UserDataHolder, Iconable {
    Project getProject();
    Language getLanguage();
    PsiElement[] getChildren();
    PsiElement getParent();
    PsiElement getFirstChild();
    String getText();
    ...
}
```

Имеет смысл отдельно акцентировать внимание на нескольких других базовых интерфейсах, играющих важную роль в процессе анализа кода в платформе IntelliJ Idea. К таковым, во-первых относится интерфейс PsiReference и PsiPolyVariantReference, которые представляют собой интерфейсы языковых объектов являющихся ссылками на какие-либо конкретные объявления в структуре программы. Первый отвечает за ссылки, однозначно разрешаемые в конкретные деревья, второй же служит для ссылок, которые в силу особенностей процесса редактирования кода в среде разработки, а так же по другим причинам, могут разрешатся в несколько элементов одновременно.

Листинг 4 - Интерфейс PsiReference

```
public interface PsiReference {
    PsiElement getElement();
    @Nullable PsiElement resolve();
    ...
}

public interface PsiPolyVariantReference extends PsiReference {
    @NotNull
    ResolveResult[] multiResolve(boolean incompleteCode);
}
```

Последним из базовых классов, на которых строится основная иерархия типов, активно используемых в системе преобразования синтаксических деревьев и семантической информации, является тип PsiType, отвечающий за представление типов(таких как примитивы, классы, массивы и.т.д.) в различных языках программирования. Более подробно преобразования типов конкретно в контексте языка *Scala* будет рассмотрены в соответствующей главе.

Листинг 5 - Класс PsiType

```
public abstract class PsiType implements PsiAnnotationOwner {
    @NotNull
    public abstract String getPresentableText();
    public boolean isAssignableFrom(@NotNull PsiType type);
    public abstract <A> A accept(@NotNull PsiTypeVisitor<A> visitor
      );
    ...
```

```
}
```

2.1.3. Преобразование простых выражений

Возможно, самой тривиальной частью преобразования синтаксических деревьев является преобразование простых выражений, таких как:

- преобразование литералов;
- преобразование нормальных вызовов;
- преобразование префиксных и инфиксных вызовов;
- преобразование блоков;
- преобразование условных выражений и циклов;
- и т.д.

В качестве примера реализации такого преобразования приводится выдержка из исходного кода конвертера, осуществляющего преобразование.

Листинг 6 - Преобразование простых выражений

```
def expression(e: ScExpression): m.Term = e match {
 case t: ScLiteral => literal(t)
 case t: ScUnitExpr =>
   m.Lit(())
     .withAttrs(toType(e.getType()))
     .setTypechecked
 case t: ScReturnStmt =>
   m.Term.Return(expression(t.expr).get)
     .withAttrs(toType(e.getType()))
     .setTypechecked
 case t: ScBlock =>
   m.Term.Block(Seq(t.statements.map(ideaToMeta(_):_*))
     .withAttrs(toType(e.getType()))
    .setTypechecked

   ...
}
```

2.1.3.1. Сопоставление с образцом

Механизм сопоставления с образцом является одним из краеугольных камней языка программирования *Scala*. В действительности, реали-

зация данного механизма сводится к набору из девяти частных случаев для реализации разной функциональности сопоставления с образцом:

— проверка типа;
— связывание значения;
— сравнение значения с литералом;
— сравнение значения с константой;
— извлечение значений из `case` классов;
— вызов метода-экстрактора;
— вызов метода-экстрактора с пустым списком аргументов;
— сопоставление с универсальным образцом;
— дизъюнкция частных случаев.

Однако синтаксические деревья, применяющиеся для декомпозиции структур при выполнении операции сопоставления с образцом, являются согласно спецификации языка, еще и при объявлении образцов в общем случае, как, например, при объявлении переменных или декомпозиции возвращаемых значений функций и методов.

Листинг 7 - Преобразование паттернов

```
def pattern(pt: patterns.ScPattern): m.Pat = pt match {
  case t: ScReferencePattern   =>  Var.Term(toTermName(t))
  case t: ScConstructorPattern=>  Extract(toTermName(t.ref), Nil,
     Seq(t.args.patterns.map(arg):_*))
  case t: ScNamingPattern      =>  Bind(Var.Term(toTermName(t)), arg
     (t.named))
  case t: ScLiteralPattern     =>  literal(t.getLiteral)
  case t: ScTuplePattern       =>  Tuple(Seq(t.patternList.get.
     patterns.map(pattern):_*))
  case t: ScWildcardPattern    =>  Wildcard()
  case t: ScCompositePattern   =>  compose(Seq(t.subpatterns : _*))
  case t: ScInfixPattern       =>  ExtractInfix(pattern(t.
     leftPattern), toTermName(t.reference), t.rightPattern.map(pt=>
     Seq(pattern(pt))).getOrElse(Nil))
  case t: ScPattern => t ?!

...
}
```

2.1.4. Преобразование объектной системы

Еще одним важным инфраструктурным элементом языка *Scala* является его функционально богатая объектная модель. В дополнение к объектной модели языка *Java*, язык *Scala* предоставляет множество дополнительных инструментов, таких как:

— примеси;
— типажи;
— селф-тайпы;
— тайп-мемберы;
— первоклассная поддержка декораторов;
— анонимные классы с ранними определениями.

Синтаксические деревья платформы IntelliJ Idea являются достаточно выразительными и позволяют проводить множество семантических операций над элементами объектной системы. Однако, синтаксически деревья платформы IntelliJ Idea в отношении объектной системы отличаются от соответствующих деревьев *scala.meta*.

Листинг 8 - Подстановка типизации

```
def template(t: p.toplevel.templates.ScExtendsBlock): m.Template =
   {
  val exprs   = t.template map (it => it.exprs.map(expression))
  val members = t.template map (it => it.members.map(ideaToMeta(_))
     )
  val early   = t.earlyDefinitions map (it => it.members.map(
     ideaToMeta(_))
  val parents = t.templateParents map (it => (it.typeElements map
     ctorParentName)
  val self    = t.selfType match {
    case Some(tpe: ptype.ScType) => m.Term.Param(Nil, m.Term.Name("
       self"), Some(toType(tpe)), None)
    case None => m.Term.Param(Nil, m.Name.Anonymous(), None, None)
  }
  val stats = (exprs, members) match {
    case (Some(exp), Some(hld)) => Some(hld ++ exp)
    case (Some(exp), None)  => Some(exp)
    case (None, Some(hld))  => Some(hld)
    case (None, None)       => None
  }
```

```
  m.Template(early, parents, self, stats)
}
```

2.1.5. Типизация выражений

Для удобства проведения различных операций над семантической информацией, ассоциированной с синтаксическими деревьями *scala.meta*, вводится дополнительная структура данных, присоединяемая к деревьям, являющимися выражениями (то есть типизируемым деревьями, не являющимися объявлениями).

Для выполнения такой ассоциации, при конвертировании деревьев платформы IntelliJ Idea, унаследованных от класса `ScExpression` дополнительно вызывается вывод типов, даже если результирующий тип не используется явно нигде далее в программе, и результат преобразовывается в соответствующий тип *scala.meta*.

Листинг 9 - Подстановка типизации

```
case t: ScReferenceExpression if t.qualifier.isDefined =>
  m.Term.Select(expression(t.qualifier.get), toTermName(t))
    .withAttrs(toType(e.getTypeWithCachedSubst))
    .setTypechecked
```

2.1.6. Кэширование результатов преобразования

Преобразования синтаксических деревьев, а также сопутствующей им семантической информации является затратной по времени операцией. В особенности, при динамическом вызове любых семантических операций в процессе исполнения метапрограмм, со значительной долей вероятности будет произведено повторное преобразование тех или иных синтаксических деревьев, как поступивших на вход программе явным образом, так и требуемых ею в процессе работы.

Учитывая тот факт, что в процессе работы метапрограммы платформой гарантируется отсутствие изменений как в самом обрабатываемом синтаксическом дереве, так и в любых других транзитивных зависимостях, было принято решение сохранять результаты преобразования синтаксических деревьев, а также сопутствующей им семантической информации в двунаправленный кэш.

Данный подход решает как проблему повторного преобразования при динамическом вызове семантических операций метапрограммой, так и позволяет не производить полное обратное преобразование имен из соответствующих структур *scala.meta* в декларации в файлах с исходным кодом из-за взаимно-однозначной природы оных, что позволяет добиться значительного ускорения всех семантических операций.

Листинг 10 - Использование кэширования

```
def toType(tp: ptype.ScType, pivot: PsiElement = null): m.Type = {
  typeCache.getOrElseUpdate(tp, {
   tp.isAliasType match {
     case Some(AliasType(ta, lower, upper)) =>
      return toTypeName(ta)
     case _ =>
   }
   tp match {
     case t: ptype.ScParameterizedType =>
  ...
}
```

2.2. Преобразование типов

Язык программирования *Scala* является строгим статически и типизированным и с возможностью локального и глобального вывода типов. Система метапрограммирования *scala.meta* позволяет работать с полным набором типов языка *Scala* и предоставляет для этого соответствующие структуры данных.

Реализация поддержки языка *Scala* и комплекса статического анализ платформы IntelliJ Idea, аналогично компилятору самого языка, включает систему вывода типов и их обработки. Для этого на платформе IntelliJ Idea создана собственная иерархия классов, отражающих как типы доступные для использования конечному программисту, такие как примитивные типы, классы, типажи, объекты, функции, экзистенциальные типы, параметризованные, и прочие типы, а также и абстрактные типы, используемые в процессе вывода типов из первой категории.

Система интеграции *scala.meta* в платформу IntelliJ Idea производит соответствующее преобразование типов для как явного использования из синтаксических деревьев обрабатываемого метапрограммой ко-

да, так и для интеграции сопутствующей семантической информации в сами деревья.

Листинг 11 - Преобразование типов

```
def toType(tp: ptype.ScType, pivot: PsiElement = null): m.Type =
  tp match {
    case t: ptype.ScParameterizedType =>
      m.Type.Apply(toType(t.designator), t.typeArguments.map(toType
          (_)).setTypechecked
    case t: ptype.api.designator.ScThisType =>
      toTypeName(t.element).setTypechecked
    case t: ptype.api.designator.ScProjectionType =>
      t.projected match {
        case tt: ptype.api.designator.ScThisType =>
          m.Type.Select(toTermName(tt.element), toTypeName(t.
              actualElement)).setTypechecked
        case _ =>
          m.Type.Project(toType(t.projected), toTypeName(t.
              actualElement)).setTypechecked
      }
    case t: ptype.api.designator.ScDesignatorType =>
      if (t.element.isSingletonType)
        toSingletonType(t.element)
      else
        toTypeName(t.element)
    case t: ptype.ScCompoundType =>
      m.Type.Compound(t.components.map(toType(_)), Seq.empty)
  ...
}
```

2.2.1. Статический вывод типов

В дальнейшем, эта операция понадобится для генерации семантической информации, в частности, в процессе генерации префиксов имен в денотациях.

2.2.2. Глубокая подстановка параметризованных типов

Ленивая и динамическая природа синтаксических деревьев и комплекса статического анализ платформы IntelliJ Idea обуславливает отсутствие необходимости сохранять как все промежуточные типы в процессе их вывода, так и подстановки параметризованных типов, что от-

личается от подхода, применяемого в платформе метапрограммирования *scala.meta*, в которой в любом дереве всегда возможно получить полностью подставленные типы в дженериках без повторного запуска вывода типов для всего выражения.

Листинг 12 - Подстановка типов

```
class Foo[+A]

class Bar extends Foo[Bar]

val a = new Bar
```

К примеру, в фрагменте кода 12 типом переменной `a` будет `Bar`, предком которого, в свою очередь будет уже специфицированный типом `Bar` класс `Foo`. В процессе анализа кода в интегрированной среде разработки данная информация является нерелевантной и может быть отброшена. Однако, для полного соответствия спецификации хранения семантической информации *scala.meta*, механизм подстановки параметризованных типов был изменен, чтобы поддерживать кэширование подстановок, произошедших в процессе вывода типа выражения. Полученный механизм достаточно прост в использовании и часто встречается при вычислении типизаций выражений в процессе конвертирования синтаксических деревьев.

Листинг 13 - Кэширование подстановок

```
...
case t: ScPostfixExpr =>
  t.withSubstitutionCaching { tp =>
    m.Term.Apply(m.Term.Select(expression(t.operand), toTermName(t.
       operation))
     .withAttrs(h.Typing.Nonrecursive(toType(tp))), Nil)
     .withAttrs(toType(tp)).setTypechecked

  }
...
```

2.2.3. η-раскрытие

Явная типизация доступна не только для выражений, имеющих конкретный тип, таких как вызовы методов или локальных перемен-

ных. Одной из ключевых особенностей, выгодно выделяющей язык программирования *Scala* из ряда прочих мультипарадигменных языков, является возможность работы с методами, как с функциями, автоматически приводя сигнатуру метода к соответствующему типу функции, и в частности, возможность передавать их в качестве аргументов другим функциям и методам.

Для поддержки такой функциональности в системе преобразования синтаксических деревьев и семантической информации осуществляется присоединение к синтаксическим деревьям, являющимся функциями и методами, типизации, согласно стандартному алгоритму η-раскрытия.

Листинг 14 - η-раскрытие

```
def toType(elem: PsiElement): m.Type = {
  psiElementTypeChache.getOrElseUpdate(elem, {
    elem match {
      case t: ScConstructor =>
        m.Type.Method(toParams(t.arguments), toType(t.getType())).
            setTypechecked
      case t: ScPrimaryConstructor =>
        m.Type.Method(t.clauses.map(convertParamClause), toType(t.
            containingClass)).setTypechecked
      case t: ScFunctionDefinition =>
        m.Type.Method(t.clauses.map(convertParamClause), toType(t.
            getTypeWithCachedSubst)).setTypechecked
      case t: ScFunction =>
        m.Type.Function(t.paramTypes.map(toType(_, t)), toType(t.
            returnType)).setTypechecked
    ...
}
```

2.2.4. Java типы

Язык программирования *Scala* является в значительной степени независимым, и даже имеет собственную стандартную библиотеку, не основывающуюся на библиотеке коллекций из *Java Runtime*. Однако, платформой для исполнения программ написанных на *Scala*, зачастую является именно *Java*, что порождает необходимость так или иначе под-

держивать операции над исходным кодом, взаимодействующим с *Java* программами.

В контексте платформы метапрограммирования *scala.meta* в общем, а также системы ее поддержки на платформе IntelliJ Idea, ситуация, при которой сама метапрограмма, либо синтаксические деревья, которые она обрабатывает, либо динамически запрашиваемая ею в процессе работы семантическая информация, будут так или иначе содержать использование классов и методов *Java*, вполне возможна и ее соответствующим образом необходимо обрабатывать.

Благодаря тому, что синтаксические деревья платформы IntelliJ Idea изначально поддерживают язык программирования *Java*, преобразование типов и методов *Java* является тривиальной задачей. Важно заметить, что имеет смысл преобразовывать только сигнатуры классов и их методы, так как трансформации деревьев *Java* в системе *scala.meta* не поддерживаются.

Листинг 15 - Java типы

```
...
case t: PsiPackage if t.getName == null =>
  m.Type.Singleton(std.rootPackageName).setTypechecked
case t: PsiPackage =>
  m.Type.Singleton(toTermName(t)).setTypechecked
case t: PsiClass =>
  m.Type.Name(t.getName).withAttrsFor(t).setTypechecked
case t: PsiMethod =>
  m.Type.Method(Seq(t.getParameterList.getParameters
    .map(Compatibility.toParameter)
    .map(i=> convertParam(i.paramInCode.get))
    .toStream),
   toType(ScTypePsiTypeBridge.toScType(t.getReturnType, t.getProject
    ))).setTypechecked
...
```

2.3. Абстракция денотаций

Ключевым отличием синтаксических деревьев и комплекса статического анализ платформы IntelliJ Idea от аналогичной системы компилятора языка программирования *Scala* является ленивое разрешение ссылок и вывод типов «на лету». Что означает, что в любой момент

времени, при обращении с к синтаксическим деревьям анализируемой программы с целью получить некую семантическую информацию, ассоциированную с ними, нет никаких гарантий, что она будет немедленно доступна. Данная особенность обуславливается тем, что в интегрированной среде разработки, в отличии от компилятора того или иного языка, накладываются дополнительные требования на рабочий процесс, например:

— возможность на лету изменять анализируемый исходный код;
— возможность производить анализ кода выходящий за рамки одной единицы компиляции;
— производительность, позволяющая выполнять анализ в интерактивном режиме.

За реализацию денотаций отвечают структуры данных, которые позволяют однозначно идентифицировать имя или ссылку, а также определить ее тип с учетом заключающего выражения.

Символ

Структура данных, служащая эквивалентом полностью квалифицированного глобального имени либо уникальный идентификатор локального имени

Префикс

Структура данных, позволяющая провести корректную типизацию при подстановке типов захватываемых из заключающего дерева

2.3.1. Генерация имен

Концепция имен является одной и фундаментальных в современных языках программирования, в том числе и в языке *Scala*. Верные имена являются однозначными идентификаторами, ссылающимися на какую-либо область в исходном коде целевой программы:

— локальные переменные и поля классов;
— классы, объекты и типажи;
— пакеты и пакетные объекты;
— тайп-алиасы и тайп-мемберы;
— параметры функций и дженериков.

Платформа метапрограммирования *scala.meta* выделяет имена в особую иерархию типов, наследующихся от базового типа Name.

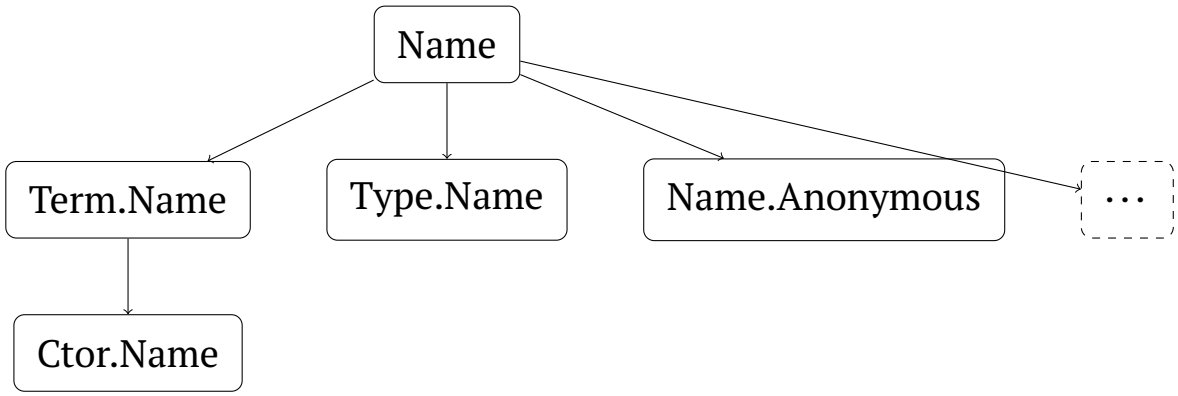


Рисунок 2 - *scala.meta*.Name

Имена в синтаксических деревьях платформы IntelliJ Idea не существуют как отдельно стоящие сущности, в отличие от деревьев *scala.meta*, а являются либо обычными полями конкретных классов, либо присоединяются к конкретным классам через типаж ScNamedElement. В зависимости от каждого отдельного случая необходимо по-разному генерировать соответствующие имена *scala.meta*.

Однако, как несложно понять, для однозначного определения конкретного места в исходном тексте программы, куда ссылается то или иное имя, одного тестового представления имени, как строкового литерала, объявленного в том или ином месте, явно недостаточно. Ярким примером могут служить одинаково называющиеся классы, расположенные в разных пакетах, или локальные переменные, скрывающие поля класса, в котором объявлена соответствующая функция.

Для решения этой проблемы, платформа метапрограммирования *scala.meta* предоставляет концепцию, основывающуюся на абстракции денотаций — такой совокупности алгоритмов и структур данных, которые в комплексе с соответствующими именами, могут предоставлять исчерпывающую семантическую информацию касательно полного разрешения имен типов, термов и прочих синтаксических деревьев.

Таким образом достигается как однозначность по идентификации символов, или по-другому — полностью квалифицированного имени, так и по типу заключающего выражения.

2.3.2. Статическое разрешение ссылок

Для синтаксических деревьев, наследующихся от таких классов платформы IntelliJ Idea, как ScReference, характерно поведение, позво-

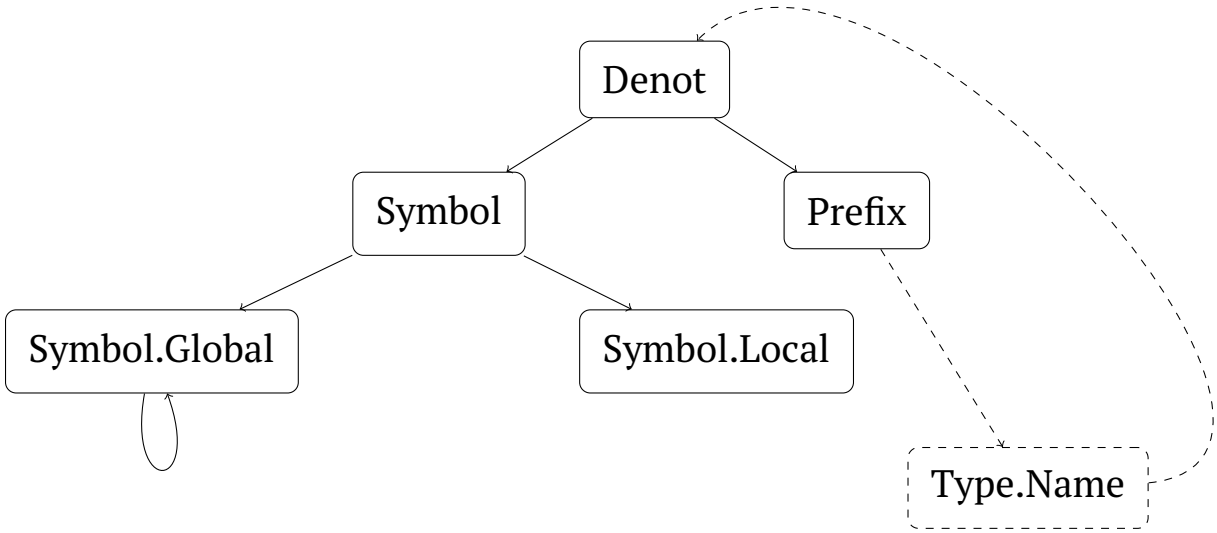


Рисунок 3 - Денотации

ляющее не разрешать ссылки до того момента, как это реально потребуется в процессе анализа исходного кода. Более того, синтаксические деревья IntelliJ Idea не предоставляют отдельных структур данных, ассоциированных с именованными выражениями, которые могли бы быть интерпретированы в качестве символов.

Система поддержки платформы метапрограммирования *scala.meta* в IntelliJ Idea ставит собой задачу статического разрешения ссылок и сохранения соответствующей семантической информации в ассоциированные деревья. Для этих целей, если в процессе преобразования синтаксических деревьев встречается дерево, являющееся однозначной или поливариантной ссылкой, происходит ее разрешение к конкретному или к некому множеству элементов, и по полученному результату генерируется соответствующая детонация.

Листинг 16 - Статическое разрешение ссылок

```
def toTermName(elem: PsiElement): m.Term.Name = elem match {
  ...
  case cr: ResolvableReferenceElement  =>
    cr.bind()  match {
      case Some(x) => try {
            toTermName(x.element)
        } catch {
```

```
        case _: SyntheticException =>
        elem.getContext match {
            case mc: ScSugarCallExpr => mkSyntheticMethodName(
              toType(mc.getBaseExpr))
            case _ =>
        }
      }
    case None => throw new \emph{Scala}MetaResolveError(cr)
  }
 case se: impl.toplevel.synthetic.SyntheticNamedElement =>
   throw new SyntheticException
 ...
```

2.3.3. Локальные символы

В случае, когда метапрограмма обращается к синтаксическим деревьям, которые не имеют глобального идентификатора, например, к таким как локальные переменные функций, все равно необходимо сгенерировать однозначно разрешаемую ссылку на данный элемент, в силу того, что при обратном разрешении ссылки из синтаксического дерева *scala.meta*, к примеру, в случае необходимости изнутри метапрограммы динамически обратиться к семантической информации, ассоциированной с данным деревом(вывести тип, получить список методов), необходимо будет в любом случае обратиться к определению данной локальной переменной в синтаксическом дереве платформы IntelliJ Idea.

Синтаксические конструкции платформы IntelliJ Idea не всегда имеют явное указание о том, что они являются локальными определениями в данном контексте, по этой причине, определять, какие деревья являются локальными определениями, а какие нет необходимо вручную.

Листинг 17 - Определение локальных символов

```
def isLocal(elem: PsiElement): Boolean = {
  elem match {
    case e: ScTypeDefinition => e.isLocal
    case e: ScBindingPattern if e.containingClass == null => true
    case pp: params.ScParameter => true
    case tp: ScTypeParam => true
    case _ => false
```

```
  }
}
```

Для решения данной задачи в системе преобразования синтаксических деревьев и семантической информации создаются особые идентификаторы локальных переменных, состоящие из полного пути к файлу, содержащим их определение, а также смещения в символах от начала самого файла. Данный подход позволяет однозначно восстановить позицию исходного элемента и быстро найти его в синтаксическом дереве указанного файла с исходным кодом.

Листинг 18 - Генерация локальных символов

```
def toLocalSymbol(elem: PsiElement): h.Symbol = {
  @tailrec
  def getFile(elem: PsiElement) =  elem.getContainingFile match {
      case _: DummyHolder => getFile(elem.getParent)
      case _ => elem.getContainingFile
  }
  val url = try {
    getFile(elem).getVirtualFile.getUrl
  } catch {
    case _: NullPointerException => "UNRESOLVED"
  }
  h.Symbol.Local(url + localSymbolDelim + elem.getTextOffset)
}
```

2.3.4. Глобальные символы

Определения, доступные из областей видимости, отличных от локальной, могут быть достижимы через ссылки на них. Подобные синтаксические деревья представляются глобальными определениями и имеют соответствующий символ. В случае с платформой метапрограммирования *scala.meta*, за это отвечает специальный тип символа — `Symbol.Global`.

Как уже замечалось ранее, синтаксические деревья платформы IntelliJ Idea не имеют обобщенной концепции символов, а вместо этого полагаются на обход деревьев «на лету» и динамическое разрешение ссылок. По данной причине необходимо производить достаточно слож-

ную генерацию символов, учитывая соотношения различных типов синтаксических деревьев платформы IntelliJ Idea.

Листинг 19 - Генерация локальных символов

```
def toSymbol(elem: PsiElement): h.Symbol = elem match {
 case _ if isLocal(elem) =>
   toLocalSymbol(elem)
 case sc: impl.toplevel.synthetic.ScSyntheticClass =>
   h.Symbol.Global(fqnameToSymbol(sc.getQualifiedName), sc.
      className, h.Signature.Type)
 case td: ScTypeDefinition if !td.qualifiedName.contains(".") =>
   h.Symbol.Global(h.Symbol.EmptyPackage, td.name, h.Signature.Type
      )
 case td: ScTemplateDefinition =>
   h.Symbol.Global(fqnameToSymbol(td.qualifiedName), td.name, h.
      Signature.Type)
 case td: ScFieldId =>
   val owner = td.nameContext match {
    case vd: ScValueDeclaration => ownerSymbol(vd)
    case vd: ScVariableDeclaration => ownerSymbol(vd)
    case other => other ?!
   }
   h.Symbol.Global(owner, td.name, h.Signature.Term)
 case td: ScFunction =>
   if (td.name == "unapply")
    toSymbol(td.containingClass)
   else {
    val jvmsig = DebuggerUtil.getFunctionJVMSignature(td).getName(
       null)
    h.Symbol.Global(ownerSymbol(td), td.name, h.Signature.Method(
       jvmsig))
   }
   ...
```

2.3.5. Поиск и генерация префиксов имен

В языках с поддержкой параметризованных типов и их вывода, и в частности в языке программирования *Scala*, недостаточно просто по единственному символу определения узнать конкретный подставленный тип. Возможна ситуация как на примере 20, где для того, чтобы вывести тип возвращаемого значения метода, необходимо подставить па-

раметризованный тип, который и будет параметризован как раз типом выражения, из которого вызывается данный метод.

Листинг 20 - Использование префиксов

```
scala> tb.typecheck(q"List(1, 2, 3).head")
res0: tb.u.Tree = immutable.this.List.apply[Int](1, 2, 3).head

scala> res0.tpe
res1: tb.u.Type = Int

scala> res0.symbol.info.finalResultType
res5: tb.u.Type = A
```

Для генерации префиксов из синтаксических деревьев платформы IntelliJ Idea при ассоциации семантической информации вычленяется заключающее выражение, выводится его тип и полученный результат соответствующим образом конвертируется в тип *scala.meta*. Важно заметить, что для разных синтаксических деревьев значащим заключающим выражением будут являться разные по типу деревья.

Листинг 21 - Генерация префиксов

```
def denot[P <: PsiElement](elem: Option[P]): h.Denotation = {
  def mprefix(elem: PsiElement, fqn: String = "") = Option(elem).
     map(
    cc => h.Prefix.Type(toType(cc))).getOrElse(fqnameToPrefix(fqn)
  )
  if (elem.isEmpty) h.Denotation.Zero
  else
    elem.get match {
      //reference has a prefix
      case cr: ScStableCodeReferenceElement if cr.qualifier.
         isDefined =>
      val results = cr.multiResolve(false)
      if (results.length > 1) {
        h.Denotation.Multi(h.Prefix.Type(m.Type.Singleton(
           toTermName(cr.qualifier.get)).setTypechecked),
          results.map(toSymbol).toList)
      } else if (results.isEmpty) {
        die(s"Failed to resolve $cr")
      } else {
        h.Denotation.Single(h.Prefix.Type(
```

```
          m.Type.Singleton(toTermName(cr.qualifier.get)).
              setTypechecked), toSymbol(cr))
        }
        case cr: ScStableCodeReferenceElement =>

        ...
}
```

2.4. Семантический контекст

Система метапрограммирования *scala.meta* позволяет не только производить синтаксические трансформации над деревьями, но и выполнять разнообразные семантические операции над ними.

Набор семантических операций, которые предоставляются системой метапрограммирования *scala.meta*, достаточно небольшой, но в то же время покрывающий весь основной набор требований, регламентирующих манипулирование семантической информацией синтаксических деревьев.

Основу данного набора операций составляет работа с определениями, ссылками и действия над типами. Например, метод `defns` позволяет получить синтаксическое дерево, являющееся определением того или иного имени, встречающегося в программе, или метод `members`, служащий для получения членов (методов, полей и прочих вложенных определений) класса.

Листинг 22 - Определение семантического контекста

```
@opaque(exclude = "dialect|domain")
trait Context {
    def dialect: Dialect
    def domain: Domain

    def typecheck(tree: Tree): Tree

    def defns(ref: Ref): Seq[Member]
    def members(tpe: Type): Seq[Member]
    def supermembers(member: Member): Seq[Member]
    def submembers(member: Member): Seq[Member]

    def isSubtype(tpe1: Type, tpe2: Type): Boolean
```

```
    def lub(tpes: Seq[Type]): Type
    def glb(tpes: Seq[Type]): Type
    def supertypes(tpe: Type): Seq[Type]
    def widen(tpe: Type): Type
    def dealias(tpe: Type): Type
}
```

Система поддержки платформы метапрограммирования *scala.meta* в IntelliJ Idea ставит собой задачу реализации корректного поведения всех вышеозначенных методов. Данные результат достигается набором преобразований синтаксических деревьев, а также операциями по динамическому разрешению ссылок и имен.

2.4.1. Обратное преобразование деревьев

В процессе выполнения метапрограммы могут вызываться семантические методы, получающие дополнительную информацию о переданных и обрабатываемых деревьях. В таких случаях часто может понадобится обращение к исходным синтаксическим деревьям, представленным в редакторе. Такими операциями могут быть, например, получение списка методов класса, получение определения функции, класса или любого другого имени, сравнение типов, и так далее.

В таком случае, система интеграции *scala.meta* производит обратное преобразование синтаксических деревьев из формата *scala.meta* в формат целевой платформы, в данном случае в формат IntelliJ Idea, и получает необходимую информацию из сохраненных ранее или динамически созданных денотаций для однозначного разрешения соответствующих имен.

Важную роль в процессе данного преобразования играет кэширование преобразованных деревьев из платформы IntelliJ Idea в соответствующие деревья *scala.meta*. В силу взаимной однозначности данного отображения, оказывается возможным не только ускорять прямое преобразование, возвращая заранее сконвертированный результат, но также немедленно находить обратное соответствие, использующихся в процессе работы метапрограммы синтаксических деревьев *scala.meta*.

Существует случай, когда использовать кэш невозможно, а такая ситуация может происходить, когда пользователь, пишущий метапрограмму, вручную генерирует синтаксические деревья и ассоциирован-

ную с ними информацию, а потом запрашивает у семантической подсистемы дополнительную информацию об указанных типах или определениях. В таком случае используется упрощенный подход, активно переиспользующий готовые системы из платформы IntelliJ Idea для построения синтаксических деревьев по тексту, в силу того, что синтаксические деревья *scala.meta* поддерживают, ровно как и синтаксические деревья платформы IntelliJ Idea, операцию быстрого преобразования в синтаксически валидный исходный код на целевом языке.

Листинг 23 - Обратное преобразование символов

```
def fromSymbol(sym: h.Symbol): PsiElement = {
  def getFqName(sym: h.Symbol): (String, Option[String]) = sym
     match {
    case h.Symbol.Global(owner, name, signature) =>
    signature match {
      case h.Signature.Type | h.Signature.Term =>
        (s"${getFqName(owner)._1}.$name", None)
      case h.Signature.Method(jvmsig) =>
        (s"${getFqName(owner)._1}.$name", Some(jvmsig))
      ...
    }
    case other => unreachable("can't get fqn of non-global symbol")
  }
  def convert: PsiElement = sym match {
    case h.Symbol.Local(id) =>
      val url::pos = id.split(localSymbolDelim).toList
      findFileByPath(url).findElementAt(pos.head.toInt)
    case h.Symbol.RootPackage => new PsiPackageImpl(PsiManager.
       getInstance(project), "")
    case h.Symbol.EmptyPackage => new PsiPackageImpl(PsiManager.
       getInstance(project), "")
    case h.Symbol.Zero => unreachable("can't map Zero symbol")
    case h.Symbol.Global(owner, name, signature) =>
    getFqName(sym) match {
        case (fqn, Some(jvmSig)) =>
        val clazz = \emph{Scala}PsiManager.instance(project).
           getCachedClass
        ...
    }
    ...
```

```
  }
  symbolCache.getOrElseUpdate(sym, convert)
}
```

Соответствующим образом, данное преобразование применяется также и для генерации конечного результата работы метапрограммы и отображении его в самом редакторе среды разработки, либо для создания специальных синтетических объявлений, которые могут не присутствовать явно в коде, но использоваться в процессе вывода типов или разрешения имен.

2.4.2. Предоставление определений имен

Одна из основных семантических операций, использующихся в процессе работы метапрограммы, является получение определения той или иной ссылки. В данной роли может выступать, к примеру, имя функции или метода, по которому пользователь хочет получить тело, или получить определение класса, объекта или типажа по его имени.

Листинг 24 - Поиск имени в дереве

```
override def defns(ref: Ref): Seq[Member] = {
  ref match {
      case pname: m.Name => getDefns(pname)
      case m.Term.Select(_, pname) => defns(pname)
      case m.Type.Select(_, pname) => defns(pname)
      case m.Type.Project(_, pname) => defns(pname)
      case m.Type.Singleton(pref) => defns(pref)
      case m.Ctor.Ref.Select(_, pname) => defns(pname)
      case m.Ctor.Ref.Project(_, pname) => defns(pname)
      case m.Ctor.Ref.Function(pname) => defns(pname)
      case _: m.Import.Selector => ???
  }
}
```

Для выполнения данной операции необходимо сначала выделить значимое имя из переданного синтаксического дерева, а после этого произвести обратное преобразование денотации, ассоциированной с ним, найти по полученной информации соответствующее синтаксическое дерево платформы IntelliJ Idea и запросить у него соответствующие поддеревья. Найденный промежуточный результат проходит при необ-

ходимости через фазу преобразования синтаксических деревьев PSI в формат *scala.meta* и возвращается метапрограмме.

Листинг 25 - Выделение членов из определения

```
def getDefns(name: m.Name): Seq[Member] = {
  val psi = name.denot.symbols.map(fromSymbol)
  val members = scala.collection.mutable.ListBuffer[Member]()
  val visitor = new \emph{Scala}ElementVisitor {
    override def visitElement(element: \emph{Scala}PsiElement) =
       element ?!
    override def visitElement(element: PsiElement) = element ?!
    override def visitTypeDefinition(typedef: ScTypeDefinition) = {
      val res = typedef match {
        case e: ScTrait => toTrait(e)
        case e: ScClass => toClass(e)
        case e: ScObject => toObject(e)
      }
    members += res
  }
  members += toFunDefn(fun)
  ...
}
```

2.4.3. Семантическая информация объектной системы

Зачастую, в процессе работы метапрограммы, требуется получить информацию об структуре объектной информации классов, переданных в качестве входных синтаксических деревьев. Среди подобных запросов, часто встречаются такие, как:

— получения списка предков класса;
— получение списка потомков класса;
— получение списка членов класса;
— поиск переопределяющих членов;
— поиск переопределенных членов.

Данная информация вычисляется при проведении линеаризации и выводе типов в платформе IntelliJ Idea, однако для правильного вывода необходимо однозначно преобразовать структуры *scala.meta* в соответствующие деревья PSI для получения полного контекста для разрешения имен и ссылок. Как и в прочих семантических методах контекста,

выполняется преобразование нужных деревьев с использованием кэширования.

В дальнейшем, полученные определения из исходного кода, отраженные в деревьях PSI передаются в реализацию семантического контекста платформы IntelliJ Idea, где из соответствующих классов динамически получаются содержащиеся в них запрашиваемые определения. В дальнейшем, полученная информация преобразуется обратно в формат *scala.meta* и возвращается метапрограмме.

Листинг 26 - Выделение членов из определения

```
override def supermembers(member : Member) : Seq[Member] = {
  val name = member match {
    case t@m.Defn.Class(_, name, _, _, _) => name
    case t@m.Defn.Object(_, name, _)    => name
    case t@m.Defn.Trait(_, name, _, _, _) => name
    case other => unreachable(s"Can't get parents of a non-class
       tree: $other")
  }
  val psi = name.denot.symbols.map(fromSymbol)
  val parents = psi.map {
    case t: ScTemplateDefinition =>
      t.supers.map(ideaToMeta(_).asInstanceOf[m.Member])
    case t: PsiClass => t.getSupers.map(ideaToMeta(_).asInstanceOf[
       m.Member]).toSeq
    case other => unreachable(s"Can't get parents of a non-class
       psi: $other")
  }
  parents.flatten
}
```

ГЛАВА 3. РЕЗУЛЬТАТЫ

3.1. Отображение деревьев

Пример кода $\Longleftrightarrow$ PSI $\Longleftrightarrow$ SMT

Листинг 27 - Объявление метода

```
 "def f(a: Int => Any)",
      Decl.Def(Nil, Term.Name("f"), Nil,
        List(List(Term.Param(Nil, Term.Name("a"),
          Some(Type.Function(List(Type.Name("Int")), Type.Name("Any
              "))), None))), Type.Name("Unit"))
```

Листинг 28 - Условное выражение

```
"if (false) 42 else 0",
Term.If(Lit.Bool(value = false), Lit.Int(42), Lit.Int(0))
```

Листинг 29 - Объявлние объекта

```
 """object Foo { def f() = 42 }
 |//start
 |Foo.f()
 """.stripMargin,
 Term.Apply(Term.Select(Term.Name("Foo"), Term.Name("f")), Nil)
```

Листинг 30 - Инстанцирование класса

```
"""class Foo(a: Int)
|//start
|new Foo(42)
""".stripMargin,
Term.New(Template(Nil, List(Term.Apply(Ctor.Ref.Name("Foo"), List(
    Lit.Int(42)))), Term.Param(Nil, Name.Anonymous(), None, None),
    None))
```

Листинг 31 - Выбрасывание исключения

```
 "throw new java.lang.RuntimeException",
 Term.Throw(Term.New(Template(Nil, List(Ctor.Ref.Select(Term.Select
     (Term.Name("java"),
 Term.Name("lang")), Ctor.Ref.Name("RuntimeException"))),
 Term.Param(Nil, Name.Anonymous(), None, None), None)))
```

Листинг 32 - Обработка исключения

```
"""
|try { () }
|catch {
|  case e: Exception => ()
|  case _ => ()
|}
|finally { () }""".stripMargin,
Term.TryWithCases(Term.Block(List(Lit.Unit())), List(Case(Pat.
    Typed(Pat.Var.Term(Term.Name("e")),
Type.Name("Exception")), None, Term.Block(List(Lit.Unit()))), Case
    (Pat.Wildcard(), None,
Term.Block(List(Lit.Unit()))), Some(Term.Block(List(Lit.Unit())))
    )
```

Листинг 33 - Цикл с условиями

```
  "for (s: Int <- Seq(1); y <- Seq(3) if y == s; z = (s, y)) {}",
  Term.For(List(
  Enumerator.Generator(Pat.Typed(Pat.Var.Term(Term.Name("s")), Type
      .Name("Int")), Term.Apply(Term.Name("Seq"), List(Lit.Int(1))))
      ,
  Enumerator.Generator(Pat.Var.Term(Term.Name("y")), Term.Apply(
      Term.Name("Seq"), List(Lit.Int(3)))),
  Enumerator.Guard(Term.ApplyInfix(Term.Name("y"), Term.Name("=="),
       Nil, List(Term.Name("s")))),
  Enumerator.Val(Pat.Var.Term(Term.Name("z")), Term.Tuple(List(Term
      .Name("s"), Term.Name("y")))),
  Term.Block(Nil))
```

Листинг 34 - Нследование

```
"""
|object A {
|trait Foo
|trait Bar
|class Baz extends A.Foo with A.Bar { Baz.super[Foo].hashCode }}
|""".stripMargin,
Defn.Object(Nil, Term.Name("A"), Ctor.Primary(Nil, Ctor.Ref.Name("
    this"), Nil),
Template(Nil, Nil, Term.Param(Nil, Name.Anonymous(), None, None),
    Some(
List(Defn.Trait(Nil, Type.Name("Foo"), Nil, Ctor.Primary(Nil, Ctor
    .Ref.Name("this"), Nil),
```

```
Template(Nil, Nil, Term.Param(Nil, Name.Anonymous(), None, None),
    None)),
Defn.Trait(Nil, Type.Name("Bar"), Nil, Ctor.Primary(Nil, Ctor.Ref.
    Name("this"), Nil),
Template(Nil, Nil, Term.Param(Nil, Name.Anonymous(), None, None),
    None)),
Defn.Class(Nil, Type.Name("Baz"), Nil, Ctor.Primary(Nil, Ctor.Ref.
    Name("this"), Nil),
Template(Nil, List(Ctor.Ref.Name("Foo"), Ctor.Ref.Name("Bar")),
Term.Param(Nil, Name.Anonymous(), None, None),
Some(List(Term.Select(Term.Super(Name.Indeterminate("Baz"),
Name.Indeterminate("Foo")), Term.Name("hashCode")))))))
```

Листинг 35 - Объявление типажа

```
"trait Foo { def foo: Int = this.hashCode }",
Defn.Trait(Nil, Type.Name("Foo"), Nil, Ctor.Primary(Nil, Ctor.Ref.
    Name("this"), Nil),
Template(Nil, Nil, Term.Param(Nil, Name.Anonymous(), None, None),
Some(Seq(Defn.Def(Nil, Term.Name("foo"), Nil, Nil, Some(Type.Name(
    "Int")),
Term.Select(Term.This(Name.Anonymous()), Term.Name("hashCode")))))
    ))
```

Листинг 36 - Сопоставление с образцом

```
"""def f = {
|case class Foo(a: Any, b: Any)
|val a = Foo(1,2)
|//start
|a match { case Foo(bar, baz) => }}
""".stripMargin,
Term.Match(Term.Name("a"), List(Case(Pat.Extract(Term.Name("Foo"),
     Nil,
List(Pat.Var.Term(Term.Name("bar")), Pat.Var.Term(Term.Name("baz")
    ))), None, Term.Block(Nil))))
```

3.1.1. Методика проверки эквивалентности деревьев

Для наиболее точного проведения тестирования эквивалентности и валидности преобразования синтаксических деревьев применяется несколько подходов. Среди них, такие как, проверка наличия полностью невалидных конструкций, таких как нулевые ссылки, параллельный обход деревьев, а также преобразование результирующих деревьев в текст и его сравнение для тестирования корректности на уровне конечного представления.

3.2. Отображение семантической информации деревьев

Пример кода ⇒ Денотация

Листинг 37 - Денотация вызова Map

```
|Defn.Val(Nil, List(Pat.Var.Term(Term.Name("a")[1])), None, Term.Apply(Term.Name("GenMapFactory")[2], List(Term.ApplyInfix(Lit.Int(1),
    Term.Name("->")[3], Nil, List(Lit.Int(2))), Term.ApplyInfix(Lit.Int(2), Term.Name("->")[4], Nil, List(Lit.Int(3))))))
|[1] Type.Singleton(Term.Name("_root_")[5])::local#temp:///src/aaa.scala*50
|[2] Type.Apply(Type.Name("GenMapFactory")[6], List(Type.Name("Map")[7]))::scala.collection.generic#GenMapFactory.apply(Lscala/
    collection/Seq;)Ljava/lang/Object;
|[3] Type.Apply(Type.Select(Term.Name("Predef")[8], Type.Name("ArrowAssoc")[9]), List(Type.Name("Int")))::scala.Predef#ArrowAssoc.->(
    Ljava/lang/Object;Ljava/lang/Object;)Lscala/Tuple2;
|[4] Type.Apply(Type.Select(Term.Name("Predef")[8], Type.Name("ArrowAssoc")[9]), List(Type.Name("Int")))::scala.Predef#ArrowAssoc.->(
    Ljava/lang/Object;Ljava/lang/Object;)Lscala/Tuple2;
|[5] 0::_root_
|[6] Type.Singleton(Term.Name("generic")[10])::scala.collection.generic#GenMapFactory
|[7] Type.Singleton(Term.Name("mutable")[11])::scala.collection.mutable#Map
|[8] Type.Singleton(Term.Name("scala")[12])::scala#Predef
|[9] Type.Name("Predef")[8]::scala.Predef#ArrowAssoc
|[10] Type.Singleton(Term.Name("collection")[13])::scala.collection.generic
|[11] Type.Singleton(Term.Name("collection")[13])::scala.collection.mutable
|[12] Type.Singleton(Term.Name("_root_")[5])::scala
|[13] Type.Singleton(Term.Name("scala")[12])::scala.collection
```

Листинг 38 - Денотация вызова List

```
|Term.Apply(Term.Select(Term.Select(Term.Select(Term.Name("scala")[1]{1}<>, Term.Name("collection")[2]{2}<>){2}<>, Term.Name("immutable
    ")[3]{3}<>){3}<>, Term.Name("List")[4]{4}<1>){5}<>, Seq(Lit.Int(42){6}<>)){7}<>
|[1] {8}::scala
|[2] {1}::scala.collection
|[3] {2}::scala.collection.immutable
|[4] {3}::scala.collection.immutable#List
|[5] {10}::scala.collection.immutable#List.apply(Lscala/collection/Seq;)Lscala/collection/immutable/List;
|[6] {1}::scala#Function1
|[7] {2}::scala.collection#Seq
|[8] {1}::scala#Nothing
|[9] {1}::scala#Int
|[10] {0}::_root_
|[11] {0}::(LOCAL)
|{1} Type.Singleton(Term.Name("scala")[1]{1}<>)
|{2} Type.Singleton(Term.Name("collection")[2]{2}<>)
|{3} Type.Singleton(Term.Name("immutable")[3]{3}<>)
|{4} Type.Singleton(Term.Name("List")[4]{4}<1>)
|{5} Type.Apply(Type.Name("Function1")[6], Seq(Type.Apply(Type.Name("Seq")[7], Seq(Type.Name("Nothing")[8])), Type.Apply(Type.Name("
    List")[4], Seq(Type.Name("Nothing")[8]))))
|{6} Type.Name("Int")[9]
|{7} Type.Apply(Type.Name("List")[4], Seq(Type.Name("Int")[9]))
|{8} Type.Singleton(Term.Name("_root_")[10]{8}<>)
|{9} Type.Method(Seq(Seq(Term.Param(Nil, Term.Name("xs")[11]{6}<>, Some(Type.Arg.Repeated(Type.Name("Int")[9])), None){6})), Type.Apply
    (Type.Name("Function1")[6], Seq(Type.Name("Int")[9], Type.Apply(Type.Name("List")[4], Seq(Type.Name("Int")[9])))))
|{10} Type.Singleton(Term.Name("List")[4]{10}<>)
|<1> Term.Name("apply")[5]{9}<>
```

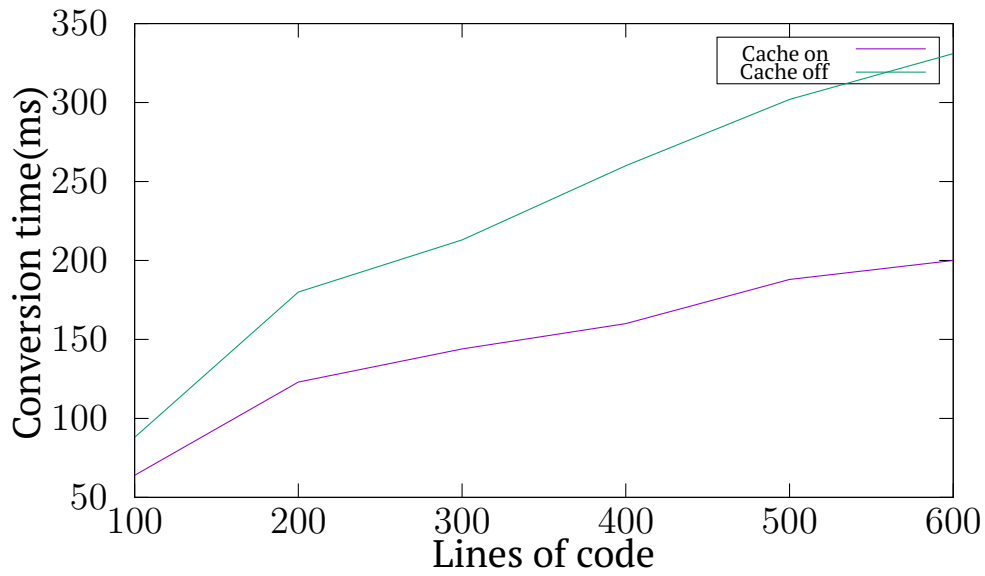
3.2.1. Методика проверки семантики деревьев

Проверка валидности семантической информации, ассоциированной с синтаксическими деревьями, производится в несколько этапов, во-первых базовая проверка на структурную валидность, такую как отсутствие нулевых ссылок и пустых денотаций, далее производится вычленение денотации из синтаксического дерева, нуждающегося в проверке, после чего у данного дерева запрашивается денотация и производится ее перевод в текстовое представление, которое впоследствии сравнивается с эталонным. Данный подход является оптимальным как с точки зрения производительности за счет относительной дешевизны операции сравнения двух строк, так и надежности по причине взаимнооднозначного отображения денотации и ее текстового представления. Более того, данный подход позволяет упростить поиск ошибок за счет простоты восприятия представления разницы строковых данных.

3.3. Методы семантического контекста

Результаты работы всех методов семантического контекста на заданных примерах кода

3.4. Производительность кэша преобразования



3.5. Применение

Полученная система является реализацией целевой платформы новой системы метапрограммирования *scala.meta* и предназначена, в первую очередь для работы по интеграции данной системы в платформу интегрированной среды разработки IntelliJ Idea.

3.5.1. IntelliJ Idea

В первую очередь система метапрограммирования *scala.meta* создавалась для абстрагирования от находящихся на более низком уровне платформ. Данная особенность позволила, в результате, прозрачным образом интегрировать данную систему в среду разработки IntelliJ Idea.

В частности, интеграция данной системы метапрограммирования с среду разработки IntelliJ Idea позволило в значительной степени облегчить и ускорить процесс разработки как самих метапрограмм, так и кода, использующего метапрограммы написанные с применением системы метапрограммирования *scala.meta*.

На момент публикации, результаты работы, проведенной в рамках данного исследования, были продемонстрированы на крупнейшей конференции, посвященной разработке программных продуктов с применением языка программирования *Scala* — «Scala Days Berlin 2016».

В демонстрации была показана функциональность, включающая в себя как непосредственно преобразования синтаксических деревьев IntelliJ Idea, так и конечная функциональность в виде возможности «на лету» получать раскрытия аннотаций, трансформирующих синтаксические деревья при помощи системы метапрограммирования *scala.meta*.

ГЛАВА 4. ЗАКЛЮЧЕНИЕ

В рамках настоящей работы была разработана система интеграции платформы метапрограммирования *scala.meta* в контексте интегрированной среды разработки IntelliJ Idea. Система поддерживает все необходимые операции по преобразованию синтаксических деревьев, их типизации, а также прочих семантических атрибутов. Полученное решение позволило разрешить такие проблемы как, вывод типов раскрытий метапрограмм и интеграции новой системы метапрограммирования в платформу анализа кода IntelliJ Idea.

Предложенный интерпретатор является одним из двух ключевых компонентов новой системы метапрограммирования языка *Scala* и входит в проект *scala.meta* по реализации обновленной системы метапрограммирования языка.

В будущем планируется продолжить работу, как над реализацией интеграции, так и над ее инфраструктурой, что позволит увеличить производительность и удобство использования обновленной системы метапрограммирования языка в среде разработки IntelliJ Idea.

Одним из последующих направлений по улучшению интеграции системы метапрограммирования *scala.meta* может являться внедрение возможности произвольных трансформаций и исправлений синтаксических деревьев в качестве правил для среды разработки, что позволит пользователям быстро и с минимальными трудозатратами создавать собственные инспекции.

Полученная система значительно упростит разработку метапрограмм для языка программирования *Scala* за счет того, что пользователи, желающие воспользоваться данной функциональностью, смогут не только тривиальным образом получать доступ ко всем аспектам работы их программы прямо в редакторе кода, но также и использовать уже все существующие в среде разработки средства анализа и коррекции кода.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1 Scala language. — URL: http://www.scala-lang.org/.

2 C++ language. — URL: http://www.cplusplus.com/.

3 E. Burmako, D. Shabalin et al "Scala.meta". — URL: http://scalameta.org.

4 *Leavenworth B. M.* Syntax Macros and Extended Translation // Commun. ACM. — New York, NY, USA, 1966. — Нояб. — Т. 9, № 11. — С. 790-793. — ISSN 0001-0782. — DOI: 10.1145/365876.365879. — URL: http://doi.acm.org/10.1145/365876.365879.

5 Macros That Work Together: Compile-time Bindings, Partial Expansion, and Definition Contexts / M. Flatt [и др.] // J. Funct. Program. — New York, NY, USA, 2012. — Март. — Т. 22, № 2. — С. 181-216. — ISSN 0956-7968. — DOI: 10.1017/S0956796812000093. — URL: http://dx.doi.org/10.1017/S0956796812000093.

6 *Burmako E.* Scala Macros: Let Our Powers Combine!: On How Rich Syntax and Static Types Work with Metaprogramming // Proceedings of the 4th Workshop on Scala. — Montpellier, France : ACM, 2013. — 3:1-3:10. — (SCALA '13). — ISBN 9781450320641. — DOI: 10.1145/2489837.2489840. — URL: http://doi.acm.org/10.1145/2489837.2489840.

7 Scala Macros, a Technical Report / E. Burmako, M. Odersky [и др.] // Third International Valentin Turchin Workshop on Metacomputation. — Citeseer. 2012.

8 *Denis Shabalin Eugene Burmako M. O.* Quasiquotes for scala: тех. отч. — Lausanne,Switzerland, 2013.

9 *Wile D. S.* Abstract syntax from concrete syntax // Proceedings of the 19th international conference on Software engineering. — ACM. 1997. — С. 472-480.

10 IntelliJ Platform SDK Documentation. — URL: http://www.jetbrains.org/intellij/sdk/docs/.