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

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЫ

«Статический и динамический вывод типов скомпилированных макросов языка программирования Scala»

Автор: Муцянко Михаил Сергеевич _____

Направление подготовки (специальность): 01.03.02 Прикладная математика и информатика

Квалификация: Бакалавр

Руководитель: Подхалюзин А.В., магистр прикл. мат. и инф. _____

К защите допустить

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

«____» _____ 20_____ г.

Санкт-Петербург, 2014 г.

Студент Муцянко М.С. **Группа** 4528 **Кафедра** компьютерных технологий **Факультет** информационных технологий и программирования

Направленность (профиль), специализация Математические модели и алгоритмы разработки программного обеспечения

Квалификационная работа выполнена с оценкой _____

Дата защиты «15» июня 2014 г.

Секретарь ГЭК _____

Листов хранения _____

Демонстрационных материалов/Чертежей хранения _____

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 совмещенной с программой проверки типов (тайпчекером) фазы компиляции и имеют такой же доступ к информации о программе, как и сам тайпчекер.

Вследствие того, что макросам предоставляются широкие возможности по манипуляции синтаксическим деревом программы, в ряде случаев возможна ситуация, когда реальный результирующий тип выражения после обработки макросом не соответствует изначально заявленному. Возможность вводить новые типы на этапе компиляции позволяет решать множество практических задач, но также привносит новую проблему, связанную с тем, что тип терма, полученного в результате работы макроса, неизвестен до этапа компиляции без раскрытия макроса.

Целью данной работы является создание метода, позволяющего выводить типы раскрытий макросов, в особенности тех, которые специфицируют тип своего результата, заранее, не прибегая к многопроходной компиляции.

ГЛАВА 1. ОБЗОР ПРЕДМЕТНОЙ ОБЛАСТИ

1.1. Макросы в языках программирования

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

— основанные на лексической токенизации;

— основанные на обработке синтаксических деревьев [3].

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

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* [4]. Синтаксические макросы позволяют преобразовывать структуру программы, используя все возможности языка, в котором они реализованы. Впоследствии 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* [5], также как и во многих других языках, являются преобразованиями, 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. В данном контексте, под раскрытием подразумевается преобразование фрагмента кода полученного из вызова метода и его аргументов.

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 между 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.

ГЛАВА 2. ПРЕДЛАГАЕМЫЙ МЕТОД

В данной главе описан предложенный метод решения задачи, поставленной в текущей работе.

2.1. Интерпретатор синтаксических деревьев

Для решения поставленной задачи предлагается написать интерпретатор типизированных абстрактных синтаксических деревьев языка *Scala*. Решение на основе интерпретации позволит разрешить все ранее поставленные задачи, а также предоставит ряд дополнительных преимуществ, детали которых будут рассмотрены ниже.

2.2. Типы синтаксических деревьев языка Scala

Прежде чем приступить к описанию деталей реализации интерпретатора, необходимо, для начала, прояснить структуру интерпретируемых данных. В дальнейшем, запись синтаксических деревьев *Scala* будет вестись в форме, указанной ниже. Рассмотрим основные типы деревьев, наиболее часто встречающихся при интерпретации:

Tree

Корень иерархии классов деревьев *Scala*. Предоставляет возможности, общие для наследников, такие как, тип(*tpe*: `Type`) и символ (*symbol*: `Symbol`).

Symbol

Символы предоставляют полную семантическую информацию о деревьях, к которым они присоединены. Символ уникален для каждого уникального объявления, такого как, например, переменная, поле класса или метод.

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

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

Apply (*fun*: Tree, *args*: List[Tree)]

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

New (*tpt*: Tree)

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

Ident (*name*: Name)

Ссылка на какой-либо объект языка с именем *name*.

Assign (*lhs*: Tree, *rhs*: Tree)

Представляет операцию присваивания. Не следует путать с методом “=”, использующимся в методах-setter’ах.

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

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

Match (*selector*: Tree, *cases*: List[CaseDef)]

Представляет операцию сопоставления значения *selector* с образцами, указанными в *cases*.

Try (*block*: Tree, *catches*: List[CaseDef , *finalizer*: Tree)]

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

Throw (*expr*: Tree)

Выбрасывает значение *expr* в качестве исключения.

Function (*vparams*: List[ValDef , *body*: Tree)]

Содержит объявление функции с телом *body* и аргументами *vparams*.

DefDef

Дерево объявления метода.

ValDef

Дерево объявления переменной. Может быть использовано как для объявления локальных переменных, так и для полей классов.

ModuleDef

Дерево объявления модулей *Scala*.

ClassDef

Дерево объявления классов или типажей (англ. *trait*).

2.3. Интерпретатор, основанный на механизме интроспекции

Одним из возможных подходов к реализации интерпретатора является реализация интерпретации лишь основных конструкций языка и возложение основной работы на механизм *runtime reflection* языка *Scala*.

Интерпретатор, основанный на механизме интроспекции, используется, в частности, в некоторых интегрированных средах разработки как основа для реализации функции «evaluate expression» в отладчике.

Данный подход позволяет упростить реализацию за счет отсутствия необходимости поддерживать такую функциональность языка, как:

— инстанцирование объектов;
— вызов методов объектов;
— обработка исключений.

Однако, подобный подход, несмотря на кажущуюся простоту, потребует реализации дополнительной функциональности в целях поддержания интероперабельности между внутренними структурами данных самого интерпретатора и нативными объектами языка.

Еще одним фундаментальным недостатком данного подхода является ограничение на множество классов, которое которое данный интерпретатор способен обрабатывать. Вследствие того, что механизм интроспекции времени исполнения языка *Scala* основан на соответствующем механизме языка *Java*, и, как следствие, может оперировать только теми классами, которые уже были скомпилированы в байткод *JVM*, данный подход бесполезен в силу того, что он явно не удовлетворяет требованию об отсутствии необходимости многопроходной компиляции либо запрещает использование в теле макроса классов и модулей, объявленных в нем же, и потенциально нарушает требование к безопасности за счет возможности исполнения произвольного кода путем прямых вызовов через механизм *Java reflection*.

2.4. Полная интерпретация

Альтернативным подходом является полная интерпретация типизированных абстрактных синтаксических деревьев языка программирования *Scala*. Подобный подход является достаточно трудоемким с точки зрения реализации, в силу необходимости полностью поддерживать де-факто стандарт языка в интерпретаторе. Сильными же сторонами такого подхода являются:

— отсутствие необходимости использовать дополнительную прослойку в виде *runtime reflection*;
— отсутствие необходимости многопроходной компиляции;
— возможность пользоваться внутри тела макроса классами и модулями, объявленными в нем же;
— отсутствие необходимости выполнять преобразования между нативными объектами языка и представлениями интерпретатора;

— изолированное исполнение кода усложняет эксплуатацию уязвимостей на этапе компиляции;
— возможность явной эмуляции небезопасных вызовов;
— отсутствие стирания типов из-за использования *runtime reflection*.

Учитывая все вышеуказанные достоинства и недостатки предоставленных методов, было принято решение использовать именно метод полной интерпретации по причине того, что лишь он удовлетворяет таким требованиям, как возможность однопроходной компиляции и изоляция среды исполнения макросов, а также предоставляет неоспоримые преимущества по сравнению с методом, основанном на механизме интроспекции времени исполнения.

2.5. Реализация интерпретатора

2.5.1. Общая схема

Реализованная модель интерпретатора, в общем случае, может быть подразделена на следующие крупные логические компоненты:

Главный цикл и основные обработчики

Данный компонент производит разбор типа входящего дерева и вызов его соответствующего обработчика, осуществляя интерпретацию всех основных конструкций языка.

Среда исполнения (`Env`)

Отвечает за хранение состояния интерпретатора в качестве пары (стек, куча). Передается в качестве аргумента в подпрограммы интерпретатора, что вкупе с неизменяемостью позволяет добиться чисто функционального дизайна языка и исключить побочные эффекты.

Значения (`trait Value`)

Классы, реализующие интерфейс `Value` реализуют конкретное поведение объектов языка, таких как функции, модули, примитивы и так далее.

Фабрика объектов и модулей

Ответственна за создание экземпляров классов и инстанцирование модулей, в том числе подлежащих эмуляции.

Символы

Набор заранее определенных символов для методов, классов и модулей, а также методов их преобразования и построения отображений, необходимых для эмуляции.

Обработка ошибок

Модуль, отвечающий за оповещение пользователя об ошибках, случающихся во время интерпретации.

Общая схема интерпретатора приведена на рисунке 1.

2.5.2. Среда исполнения

В данном интерпретаторе реализована стандартная модель памяти, представляющая из себя пару «стек+куча». В стеке, подобно стеку *JVM*, хранятся стековые кадры, соответствующие локальным областям памяти для функций. В куче хранятся динамически выделенные объекты.

Кадр стека в интерпретаторе является отображением из множества символов, однозначно определяющих переменную, метод, класс или модуль, в множество ссылок на значения. При каждом вызове функции на стеке выделяется новый кадр и происходит захват стекового кадра из внешней области видимости для возможности ссылаться на свободные переменные в контексте вызова — поддержки замыканий. Метод, перед вызовом, также расширяет

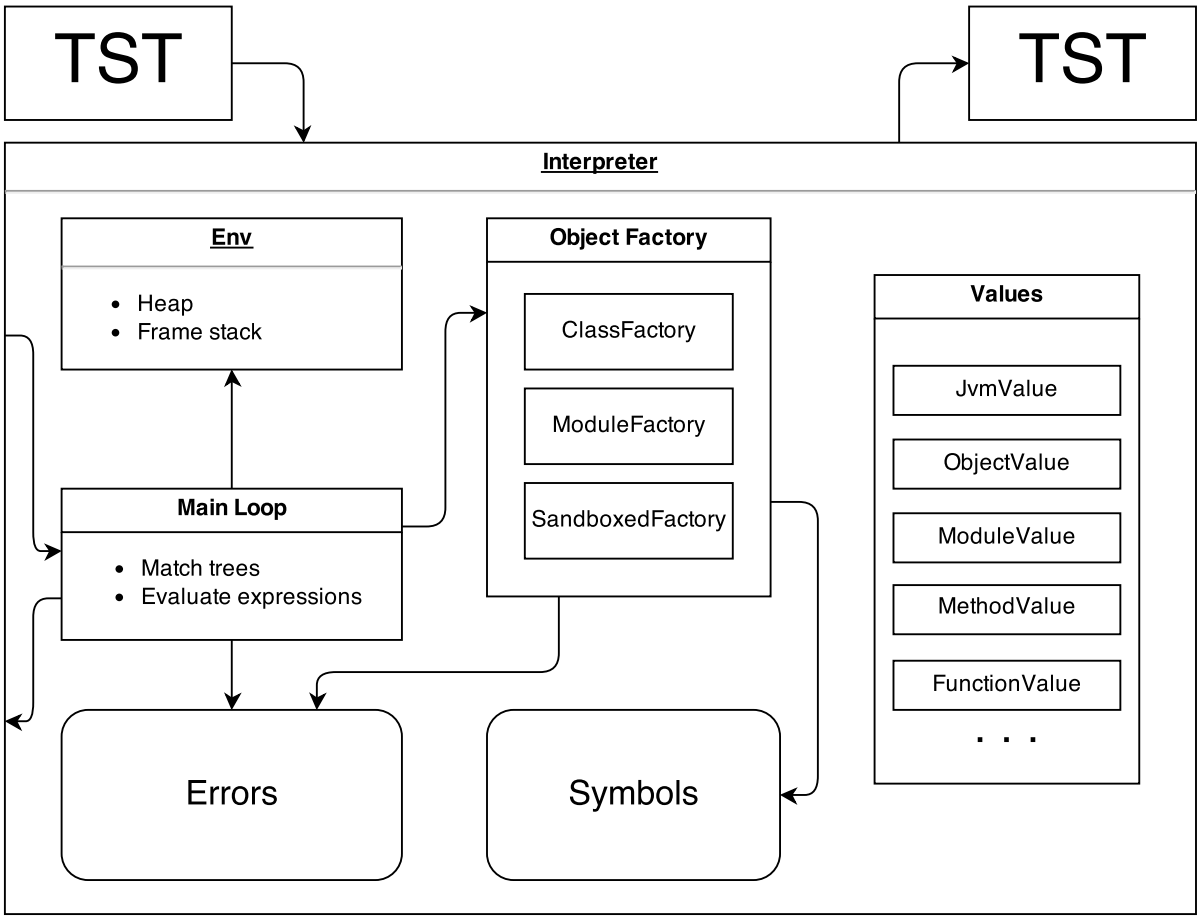

Рисунок 1 – Общая схема интерпретатора

лексическую область видимости собственным символом для поддержки рекурсивных вызовов.

Куча содержит сами значения, ссылки на которые хранятся на стеке. Значения в свою очередь подразделяются на примитивы, отражающие собой примитивы *JVM*, и объекты, содержащие поля классов. API среды исполнения предусматривает операцию импорта кучи из другой среды.

Стоит заметить, что среда исполнения не является глобальным состоянием интерпретатора, а передается в качестве аргумента в каждую из подпрограмм, где осуществляется работа с информацией, которую она предоставляет. Вкупе с реализацией стека и кучи на основе неизменяемых структурах данных, это позволяет добиться отсутствия нежелательных побочных эффектов и, как следствие, более надежной и предсказуемой работы интерпретатора. Кроме того, использование неизменяемых структур данных упрощает реализацию за счет отсутствия необходимости реализовать дополнительную логику для сброса кадра при выходе из функции и лексических областей видимости при выходе из блока кода.

На листинге 3 представлен интерфейс среды исполнения. В целях повышения читаемости, реализация методов опущена.

Листинг 3 – Интерфейс среды исполнения

```
case class Env(stack: FrameStack, heap: Heap) {
  def lookup(sym: Symbol): Result
  def extend(sym: Symbol, value: Value): Env
  def extend(obj: Value, field: Symbol, value: Value): Env
  def extend(obj: Value, newFields: Map[Symbol, Value]): Env
  def extend(v: Value, s: Slot): Env
  def extendHeap(other: Env): Env
  def extend(v: Value, a: Any): Env
  def pushFrame(other: Env)
}
```

2.5.3. Интерфейс значений интерпретатора

Все объекты представления интерпретатора имеют общий интерфейс, предоставляющий общие методы для работы с языковыми объектами *Scala*. Представление значений в интерпретаторе должно предоставлять обобщенный

метод взаимодействия, характерный для всех объектов-значений. Например, операции выбора поля или метода, применения, проверки и приведения типа.

Помимо этого, значения должны иметь интерфейс, предоставляющий возможность их приведения к типам данных *Java*, копирования(в случае передачи по значению) и инициализации(в случае модулей).

Все вышеозначенные требования привели к реализации интерфейса для значений интерпретатора, предоставленного на листинге 4, реализуют который такие классы как `JvmValue`, `ObjectValue`, `FunctionValue` и другие.

Листинг 4 – Интерфейс значения интерпретатора

```
sealed trait Value {
   def reify(env: Env): JvmResult
   def select(member: Symbol, env: Env, static: Boolean = false):
      Result
   def apply(args: List[Value], env: Env): Result
   def branch[T](then1: Env => T, else1: Env => T, env: Env): T
   def typeTest(tpe: Type, env: Env): Result
   def typeCast(tpe: Type, env: Env): Result
   def copy(env: Env): Result
   def init(env: Env): Result
   val tpt: Type
 }
```

Полное дерево классов значений интерпретатора предоставлено на рисунке 2. Элементами, обозначенными пунктирными линиями, являются эмулируемые классы и методы, которые реализованы отдельно и поведение которых зависит от конкретных методов, которые они эмулируют. Более подробно реализация данного подхода описана в главе 2.5.7.

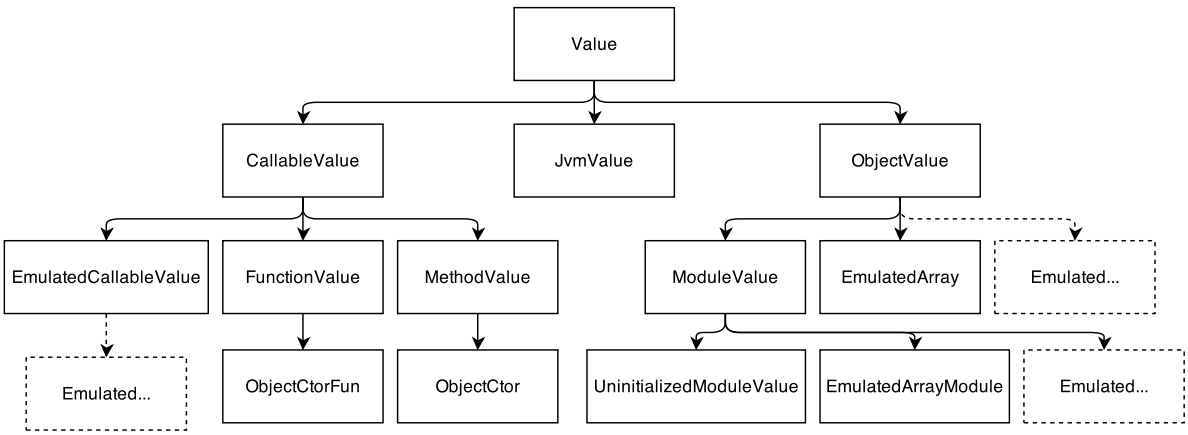

Рисунок 2 – Диаграмма наследования значений интерпретатора

2.5.4. Главный цикл и основные обработчики

В данной реализации интерпретатора делается попытка придерживаться функционального стиля программирования. В соответствии с этим, в основе интерпретатора лежит не использование паттерна «visitor», а применение метода сопоставления с образцом. Вкупе с использованием квазицитат в качестве паттернов, значительно улучшается читаемость кода и облегчается его сопровождение.

Основной метод интерпретатора представляет собой единую конструкцию, выполняющую сопоставление аргумента — дерева с образцами, представляющими собой основные типы синтаксических деревьев компилятора *Scala*. На листинге 5 предоставлен фрагмент кода главного метода интерпретатора.

Обработчики, вызываемые из главного метода, отвечают за интерпретацию следующих языковых конструкций:

— обращение к переменной;
— создание объекта;
— доступ к литералу;
— сопоставление типа значения;
— вызов метода или функции;
— операция присваивания;
— возврат значения из метода;
— выбрасывание исключения;
— условный переход;
— сопоставление с образцом;
— блок обработки исключений;
— цикл с предусловием;
— цикл с постусловием;
— объявление метода;
— объявление переменной;
— объявление модуля;
— объявление класса.

Все обработчики имеют в качестве типа возвращаемого значения пару `(Value, Env)`, что соответствует семантике языка *Scala*, в котором все конструкции, кроме объявлений возвращают значение.

Листинг 5 – Фрагмент метода, осуществляющего разбор дерева

```
def eval(tree: Tree, env: Env): Result = tree match {
  case q"${value: Value}"          => (value, env)
  case EmptyTree                   => eval(q"()", env)
  case Literal(_)                  => evalLiteral(tree, env)
  case New(_)                      => Value.instantiate(tree.tpe,
     env)
  case Ident(_)                    => env.lookup(tree.symbol)
  case s: Super                    => eval(s.qual, env)
  case q"$qual.$_"                 => evalSelect(qual, tree.symbol,
     env)
  case q"$_.this"                  => env.lookup(tree.symbol)
  case q"$expr.isInstanceOf[$tpt]" => evalTypeTest(expr, tpt.tpe,
     env)
  case q"$expr.asInstanceOf[$tpt]" => evalTypeCast(expr, tpt.tpe,
     env)
  case Apply(expr, args)           => evalApply(expr, args, env)
  ...
}
```

2.5.4.1. Обращение к переменной

Обращение к переменной происходит посредством поиска в стеке средой исполнения значения, соответствующего символу, ассоциированному с переменной. В самом деле, аналогичным образом получаются ссылки на «this» — указатель на текущий экземпляр класса. Подробнее об этом будет рассказано в главе 2.1.4.5. Отдельно обрабатывается ситуация, когда запрашиваемому символу соответствует вызываемый объект с нулевым списком списков агрументов. В таком случае, в соответствии со стандартом языка, такой объект необходимо вызвать, а его результат передать в качестве возвращаемого значения обработчика.

2.5.4.2. Создание объекта

При создании экземпляра класса, происходит передача управления в фабрику объектов. При этом необходимо заметить, что интерпретация дерева `New` не являет собой вызов конструктора. Полностью же, создание объекта в записи в виде синтаксического дерева может выглядеть так: `Apply(Select(New(tpe), termNames.CONSTRUCTOR), _)`

2.5.4.3. Литералы

Литералы интерпретируются путем оборачивания константы, которая содержится в структуре дерева `Literal`, в экземпляр класса `JvmValue`. При

этом происходит сохранение типа значения для дальнейшего использования в таких операция, как, например, проверка или приведение типа.

2.5.4.4. Вызов метода или функции

Обработка дерева вызова метода или функции достаточно проста в своей реализации за счет того, что большая часть логики исполнения вынесена в соответствующие классы `FunctionValue` и `MethodValue`. В самом же обработчике интерпретируется левое поддерево дерева `Apply` — получение вызываемого объекта, и правое — набор аргументов, после чего управление передается в полученный вызываемый объект. В этом же обработчике происходит перехват возвращаемого результата в случае, когда внутри при интерпретации тела вызываемого объекта была встречена конструкция `return`. Важно заметить, что выражение `return` в языке `Scala` относится к ближайшему заключающему методу, что вынуждает сбрасывать стековые кадры до тех пор, пока контекст исполнения не будет являться именно методом, а не функцией.

Листинг 6 – Обработчик дерева Apply

```
def evalApply(expr: Tree, args: List[Tree], env: Env): Result = {
  val (vexpr, env1) = eval(expr, env)
  val (vargs, env2) = eval(args, env1)
  try {
    vexpr.apply(vargs, env2)
  } catch {case e@ReturnException(ret) => if
     (vexpr.isInstanceOf[MethodValue]) ret else throw e}
}
```

2.5.4.5. Присваивание

Присваивание интерпретируется тривиально — сначала получается ссылка на объект, соответствующий левому поддереву, в случае, когда левое поддерево является деревом выборки из объекта; интерпретируется правое поддерево, и соответствующим результатом обновляется возвращаемая среда исполнения.

2.5.4.6. Возврат значения и выбрасывание исключения

Возврат значения из метода и операция выбрасывания исключения интерпретируются с использованием схожего подхода. Вначале вычисляется значение поддерева, которое является аргументом операции возврата или выбра-

сывания исключения. Полученный результат далее оборачивается в экземпляр специального класса `ReturnException` или `WrappedException` для операции возврата и выбрасывания исключения соответственно и выбрасывается стандартным механизмом языка *Scala*. В результате происходит раскрутка стека самого интерпретатора до тех пор, пока не будет встречен вызов обработчика блока `try catch` или вызов вызываемого объекта. Благодаря тому, что в результате, обернутом в специальный класс, содержится также среда исполнения, сохраненная в момент выбрасывания исключения или выполнения инструкции возврата, становится возможным восстановить кучу, и, как следствие, само перехваченное значение из результата.

2.5.4.7. Циклы

Циклы с предусловием являются вместе с циклами с постусловием являются единственными циклами в языке *Scala* с точки зрения императивной парадигмы программирования в силу того, что циклы `for` представляются в языке *Scala* методами, а не отдельными языковыми конструкциями. Цикл с предусловием в интерпретаторе, как и многие другие элементы, реализован в функциональном стиле через применение хвостовой рекурсии. Интерпретация цикла с постусловием преобразуется в процессе в однократную интерпретацию тела цикла и последующий вызов обработчика цикла с предусловием.

Листинг 7 – Обработчик цикла с предусловием

```
@tailrec
private def evalWhile(cond: Tree, body: Tree, env: Env): Result =
    {
  val (vcond, env1) = eval(cond, env)
  val (jcond, env2) = vcond.reify(env1)
  jcond match {
    case true =>
      val (_, env3) = eval(body, env2)
      evalWhile(cond, body, env3)
    case false =>
      Value.reflect((), env2)
    case _ =>
      IllegalState(jcond)
  }
}
```

2.5.4.8. Сопоставление с образцом

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

Все вышеозначенные конструкции являются примитивными в терминах текущей реализации интерпретатора и передаются в соответствующие им обработчики. Также, отдельно от самого паттерна интерпретируется `guard` выражение, по результату которого уже принимается решение о необходимости интерпретировать выражение, связанное с телом паттерна.

2.5.4.9. Инициализация переменной

Инициализация переменных в языке программирования *Scala* разделяются компилятором на два этапа:

а) инициализация значением по умолчанию;
б) инициализация значением из правого поддерева.

В соответствие с данной семантикой происходит интерпретация объявления и инициализации переменных. Важно заметить, что при интерпретации переменных с модификатором `lazy` второй фазы не происходит, а инициализация значением из правого поддерева осуществляется при вызове соответствующего метода, при первом обращении к переменной. Также, помимо выбора метода инициализации переменной, необходимо определить, в каком контексте она была объявлена — как локальная переменная, или как поле класса. В зависимости от этого происходит либо расширение среды исполнения локальной переменной, либо полем в экземпляре соответствующего класса.

2.5.5. Методы и функции

Язык программирования *Scala* обладает широкой поддержкой функциональной парадигмы программирования, позволяя использовать такие возможности, как например лямбда-функции, частичное применение, множественные списки параметров, замыкания, функции высшего порядка, передача вызываемых значений в качестве аргументов. Все эти возможности реализованы в данной модели интерпретатора.

Все вызываемые значения в текущей реализации реализуют интерфейс `CallableValue`. Среди них класс `MethodValue`, соответствующий методам языка *Scala*. Метод, в отличие от функции, имеет символ, при использовании которого, при вызове метода `apply`, лениво загружается его тело. После чего создается экземпляр `FunctionValue` и от него уже вызывается метод `apply`, предварительно расширив среду выполнения ссылкой на вызываемый метод для возможности осуществления рекурсивных вызовов. Кроме того, при создании экземпляра `MethodValue`, осуществляется захват среды исполнения из контекста объявления, это позволяет осуществить замыкание, в дальнейшем, захваченная среда передается и в созданное функциональное значение.

При вызове соответствующего функционального значения первым шагом осуществляется добавление на верхушку стека среды исполнения нового кадра, причем захват переменных происходит автоматически. Далее определяется метод вызова:

— вызов с единственным списком параметров;
— вызов с множественным списком параметров;
— вызов с пустым списком списков параметров.

В первом и последнем случаях производится расширение среды исполнения аргументами, и интерпретация тела функции. В противном случае возвращается частично примененная функция с подставленными аргументами из головы списка списков параметров.

2.5.6. Объектная модель

В данном разделе, равно как и в остальных, функциональность, связанная с системой типов языка *Scala* не описывается, в силу того, что в интерпретатор поступают уже типизированные синтаксические деревья, и вся работа по выведению и проверке типов априори считается выполненной.

Еще однми 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.

2.5.6.1. Конструктор класса

Основная работа по инстанцированию класса выполняется, как и в условиях выполнения в *JVM*, на этапе вызова конструктора. Конструктор является методом класса с фиксированным именем: для классов и объектов — `<init>`, для типажей — `$init$`.

Конструктор в контексте интерпретатора — специальный класс, реализующий интерфейс *Callable*, и возвращающийся при выборе из экземпляра класса соответствующего символа. Задача метода, оборачивающего конструктор — выполнить необходимые предварительные преобразования, необходимость которых вызвана тем, что синтаксические деревья, подаваемые компилятором, часто не содержат достаточно информации, необходимой для их прямой интерпретации, в силу того, что часть фаз компиляции, необходимых для построения полного синтаксического дерева, выполняется после фазы *typer*, частью которой является интерпретация макросов. К необходимым предварительным преобразования относятся:

— экстракция и интерпретация «ранних определений»;
— сопоставление аргументов конструктора полям класса;
— загрузка тела конструктора.

После выполнения всех необходимых предварительных действий, управление передается функции, содержащей тело конструктора. Порядок действий, выполняемых при вызове конструктора определен стандартом языка *Scala* делится на следующие фазы:

а) вызов конструктора суперкласса;
б) инициализация примесей в порядке, обратном линеаризации;

в) интерпретация последовательности выражений тела конструктора.

Также, конструктор в интерпретаторе возвращает значение с фиксированным типом — результат, соответствующий экземпляру класса, инициализируемого в конструкторе. В остальном, семантика вызова остается аналогичной прочим вызываемым объектам интерпретатора, к примеру, сохраняется возможность использовать конструкторы с множественными списками параметров.

2.5.6.2. Выбор полей и методов

Возможность выбора полей и методов из экземпляров классов является одной из базовых возможностей объектной модели, что делает ее поддержку необходимой при интерпретации.

Выбор поля или метода в терминах синтаксических деревьев языка *Scala* представляется деревом типа `Select(qual, name)`, где в качестве поля `qual` передается дерево объекта выбора, а `name` — имя поля или метода. Помимо этого, дереву `Select` также сопоставлен символ, однозначно определяющий поле или метод, выбираемый из объекта.

При выборе поля или метода, в качестве левого поддерева могут выступать любые деревья, интерпретация которых возвращает значение. К примеру, вызов метода или обращение к локальной переменной.

На листинге 8 представлен алгоритм выбора метода из значения.

Иными словами, при выборе метода могут возникать три случая:

Статический выбор неабстрактного метода

Применяется выборе методов с использованием ключевого слова `super`. Примером использования может служить вызов конструктора предка из конструктора потомка. В данном случае, статически определяет символ необходимого метода, и нет необходимости проводить дополнительные вычисления.

Статический выбор абстрактного метода

Данный частный случай покрывает функциональность языка *Scala*, известной как *abstract overrides* или *stackable traits*. Подобная ситуация возникает, когда выбираемый из предка метод является абстрактным. В таком случае необходимо в списке линеаризации найти первого предка с не абстрактным методом, переопределяющим данный, и выбрать из него метод с совпадающей сигнатурой.

Листинг 8 – Алгоритм выбора символа метода

```
1: if isStatic(member) ∧ ¬isAbstract(member) then return member
2: else if isStatic(member) ∧ isAbstract(member) then
3:     for baseClass ← baseClasses do
4:         for m ← methods(baseClass) do
5:             if typeSignature(m) = typeSignature(member)∧¬isAbstract(m)
   then
6:                 return m
7:             end if
8:         end for
9:     end for
10: else
11:     for m ← methods(currentClass) do
12:         if typeSignature(m) = typeSignature(member) then
13:             return m
14:         end if
15:     end for
16: end if
```

Обычный виртуальный вызов

Все остальные вызовы. В данном случае применяется стандартная логика вызова виртуальных методов — выбирается соответствующий метод из сигнатуры класса, сохраненной при его создании.

2.5.7. Фабрика объектов и эмулируемые классы и модули

Не для всех элементов языка *Scala* существует возможность их интерпретации в виде представления синтаксических деревьев. *Scala*, хоть и являясь в основном *self-hosted* языком, все же осуществляет взаимодействие со средой исполнения *Java*, на которой базируется.

Примером такого поведения могут служить класс `Array`, предоставляющий функциональность массивов *Java*. В самом деле, если посмотреть на исходный код данного класса в стандартной библиотеке *Scala*, то можно увидеть, что вместо реализации методов стоят заглушки, что делает работу с массивами на основе синтаксических деревьев невозможной.

Другим примером могут являться примитивные типы языка, которые расширяют функциональность примитивных типов *Java*. На момент написания, в число прямых потомков класса `AnyVal`, представляющих собой примитивные типы, входило девять классов. Каждый такой класс имеет свой на-

бор методов, реализующих конкретное поведение. Для класса `Int`, количество таких методов составляет 110. Реализации этих методов также отсутствует.

Еще одну проблему вызывает поставленная задача о необходимости создания изолированной среды исполнения с минимизацией побочных эффектов. Действительно, полностью запрещать пользователю взаимодействовать со средой было бы неразумно, так как в макросах может использоваться, например, информация о переменных среды, которая получается получается при вызове метода `System.getenv()` из *Java runtime*.

Все вышеозначенные проблемы предполагают создание решения, способного эмулировать поведение соответствующих объектов или заданной функциональности. Дополнительным требованием является возможность явно контролировать доступность определенных методов и классов.

Для решения данной проблемы вводится дополнительные сущности интерпретатора — фабрика классов и модулей, и фабрика методов. В ответственность первой входит создание классов и модулей, поведение которых требуется эмулировать в интерпретаторе. Фабрика методов же предоставляет эмуляцию поведения конкретных методов, таких как, например, операторы примитивных типов или встроенные методы *Java*, такие как `hashCode` и `equals`.

Обе фабрики представляют собой, в общем случае, отображения $Symbol \to Value$. Используя свойство уникальности и однозначности символов, можно реализовать такую фабрику на основе хэш-таблиц, где в качестве ключей выступают соответствующие символы классов, модулей или методов, а в качестве значений — порождающие функции. Причем, если заданный символ в таблице отсутствует, необходимо вернуться к поведению по умолчанию и создать объект на основе имеющихся синтаксических деревьев.

Благодаря интерфейсу `Value`, определение эмулируемых классов представляет собой тривиальную задачу — достаточно в новом эмулируемом классе переопределить поведение метода `Value.select()` таким образом, чтобы при выборе методов, возвращать такое вызываемое значение, проведение которого реализовано в коде самого эмулируемого объекта. Для простоты понимания, на листинге 9 приведен фрагмент кода эмулируемого класса `Array`.

Подход к эмуляции методов аналогичен подходу для классов, с отличием в том, что переопределять необходимо не поведение метода `Value.select()`, а метода `Value.apply()`. Для удобства использования

Листинг 9 – Фрагмент эмулируемого класса Array

```
class EmulatedArray(tpe: Type) extends TypedValue(tpe) {
  var data: Array[Value] = null
  def constructFrom(v: Array[Value]): EmulatedArray = {data = v;
     this}
  override def select(member: Symbol, env: Env, static: Boolean):
     (Value, Env) = {
    if (member.isConstructor)
      ec((args, env) => constructFrom(tpe.typeArgs.head,
         args.head.reify(env), env), nullary = false, env)
    else if (member == Array_apply)
      ec((args, env) => (data(args.head.reify(env)), env), nullary
         = false, env)
    else if (member == Array_update) {
      ec((args, env) => {
         data(args.head.reify(env)) = args.tail.head
         Value.reflect((), env)
       } , nullary = false, env)
    } else if (member == Array_length)
      ec((args, env) => Value.reflect(data.length, env),nullary =
         true, env)
      ...
  }
  ...
}
```

был введен класс `EmulatedCallableValue`, позволяющий упростить создание вызываемых объектов путем конструирования их тела напрямую из лямбда-функции.

Листинг 10 – класс EmulatedCallableValue

```
case class EmulatedCallableValue(f: (List[Value], Env) => Result)
   extends CallableValue {
  override def apply(args: List[Value], env: Env) = f(args, env)
}
```

Для эмуляции методов примитивов используется иной подход. Действительно, вручную реализовывать несколько сотен методов примитивов является исключительно контрпродуктивным решением. В итоге был выбран способ, основанный на использовании библиотеки времени выполнения языка *Scala*. Данная библиотека предоставляет класс `BoxesRunTime`, который выполняет все необходимые приведения и разбор типов примитивов при осуществлении операций над ними. Отсутствие необходимости вручную разбирать типы со-

кращает размер требуемого отображения до 23, по числу операторов над примитивами языка *Scala*. Сам же выбор и вызов методов класса `BoxesRunTime` осуществляется при помощи механизма интроспекции времени выполнения.

ГЛАВА 3. РЕЗУЛЬТАТЫ

В результате выполнения данной работы был написан интерпретатор типизированных абстрактных синтаксических деревьев языка *Scala*, обладающий возможностью интерпретировать следующие языковые конструкции. Для простоты понимания, все примеры в данном разделе возвращают фиксированный результат — число 42.

3.1. Интерпретируемые конструкции языка

3.1.1. Примитивы и операции над ними

Листинг 11 – Литерал

```
42
```

Листинг 12 – Объявление переменной

```
var a = 42
a
```

Листинг 13 – Изменение переменной

```
var a = 0
a = 42
a
```

Листинг 14 – Объявление константы

```
val a = 42
a
```

Листинг 15 – Операция над константами

```
val a = 40
val b = 2
a+b
```

Листинг 16 – Ленивая инициализация

```
var a = 999
lazy val b = a
a = 42
b
```

3.1.2. Методы и функции

Листинг 17 – Метод с пустым списком параметров

```
def f = 42
f
```

Листинг 18 – Метод с единственным списком параметров

```
def f(a: Int,b: Int) = a+b
f(40,2)
```

Листинг 19 – Метод с множественным списком параметров

```
def f(a:Int)(b:Int) = a+b
f(40)(2)
```

Листинг 20 – Метод как аргумент

```
def f(g: => Int) = g
def g(a: Int) = a + 2
f(g(40))
```

Листинг 21 – Лямбда-функция с пустым аргументом

```
{ () => 42 }
```

Листинг 22 – Сохранение лямбда-функции

```
val f = {a: Int => a+2}
f(40)
```

Листинг 23 – Лямбда-функция в качестве аргумента

```
def f(g: Int => Int, a: Int) = g(a) + 2
f(x => x + 30, 10)
```

Листинг 24 – Замыкание метода

```
val v = 2
def f(a: Int) = a+v
f(40)
```

Листинг 25 – Замыкание лямбда-функции

```
val v = 40
def g(f: Int, Int => Int, i: Int) = f(i,i)
g((a: Int, b: Int) => a+b+v, 1)
```

Листинг 26 – Частичное применение

```
def f(a:Int)(b:Int) = a+b
val g = f(40)(_)
g(2)
```

Листинг 27 – Возвращение значения из метода

```
def f: Int = {
   return 42
   999
}
f
```

Листинг 28 – Возвращение значения из лямбда-функции

```
def f(g: => Int) = {g; 888}
def k:Int = {f({return 42; 777}); 999}
k
```

3.1.3. Управляющие конструкции

Листинг 29 – Условный переход

```
val a = 1
if (a == 1)
  if (a+1 == 2)
    if (a+2 == 3)
      42
    else 888
  else 777
else 666
```

Листинг 30 – Цикл с предусловием

```
var a = 0
while (a < 42)
  a = a + 1
a
```

Листинг 31 – Цикл с постусловием

```
var a = 0
do { a = a + 1}
while (a < 42)
a
```

Листинг 32 – Сопоставление с образцом

```
class A
class B extends A
val v1 = new B match {
  case _: A => 20
  case _: B => 999
  case _ => 888
}
val v2 = new B match {
  case _: B => 22
  case _: A => 999
  case _ => 888
}
v1+v2
```

3.1.4. Объектная модель

Листинг 33 – Модуль с изменяемыми полями

```
object A {var v = 0}
A.v = 42
A.v
```

Листинг 34 – Замыкание модуля

```
val a = 20
object A {
  val b = 2
  object B {def foo(c: Int) = a+b+c}
}
A.B.foo(20)
```

Листинг 35 – Ленивая инициализация модуля

```
var a = 40
object A { a = 2; val b = 80}
val c = a
A.b-c+a
```

Листинг 36 – Ленивая инициализация полей модуля

```
object A {
  var v = 100
  lazy val a = v
  v = 42
}
A.a
```

Листинг 37 – Конструктор со множественным списком параметров

```
class A(a:Int)(b:Int){def f = a+b}
new A(40)(2).f
```

Листинг 38 – Множественное наследование

```
trait A{def a = 10}
trait B{def b = 30}
trait C{def c = 2}
class F extends A with B with C {def f = a+b+c}
(new F).f
```

Листинг 39 – Виртуальные вызовы

```
class A { def f() = 100 }
class B extends A { override def f() = 42 }
val b: A = new B
b.f()
```

Листинг 40 – Виртуальный вызов перегруженного метода

```
class A{ def f = 100; def f(a: Int) = a + 100}
class B extends A{ override def f = 999; override def f(a: Int) =
   a}
(new B).f(42)
```

Листинг 41 – Вызов метода предка

```
class A{def f() = 100}
class B extends A{override def f() = super.f()+42}
(new B).f()
```

Листинг 42 – Abstract override

```
trait Foo { def f: Int }
trait M extends Foo { abstract override def f = 22 + super.f }
class FooImpl1 extends Foo { override def f = 20 }
class FooImpl2 extends FooImpl1 with M
(new FooImpl2).f
```

Листинг 43 – Анонимный класс

```
trait F {val x: Int}
val v = new F {val x = 42}
v.x
```

Листинг 44 – Класс с ранними определениями

```
trait C { val x = 100 }
class D extends { override val x = 42 } with C
(new D).x
```

Листинг 45 – Анонимный класс с ранними определениями

```
trait F { val x: Int }
val v = new { val x = 42 } with F
v.x
```

Листинг 46 – Self-type

```
trait A {def f = 999}
trait B extends A {override def f = 22}
trait K {def k = 888}
trait M extends K {override def k = 19}
class C {
  self: A with K=>
  def g = 1 + f + k
}
(new C with B with M).g
```

Листинг 47 – Замыкание класса

```
class A(a: Int) {
  class Inner {def f(v: Int) = a + v}
  def g = (new Inner).f(22)
}
new A(20).g
```

3.1.5. Исключения

Листинг 48 – Выбрасывание и поимка исключения

```
class A extends Throwable
try {
  throw new A
} catch {
  case _:A       => 42
  case _:Throwable => 999
}
```

Листинг 49 – Исключение внутри вложенного вызова

```
class A extends Throwable
def f = {throw new A; 999}
def g = {f; 777}
try { g } catch {
  case _:A       => 42
  case _:Throwable => 888
}
```

3.1.6. Эмулируемые объекты

Листинг 50 – Чтение элемента массива

```
val a = new Array[Int](1)
a(0) + 42
```

Листинг 51 – Запись элемента массива

```
val a = new Array[Int](1)
a(0) = 42
a(0)
```

3.2. Применение

3.2.1. Интеграция в макросистему языка Scala

Интерпретатор синтаксических деревьев *Scala* является частью проекта *Palladium*[1], ставящим своей целью облегчить использование макросов языка *Scala* и расширить их возможности. Проект *Palladium* является развитем проекта *Kepler*[2], привнесшего макросистему в язык *Scala*.

Среди глобальных задач проекта существуют такие как:

— доступное API интроспекции, не привязанное к API компилятора;
— механизм квазицитат для упрощенной манипуляции синтаксическими деревьями;
— интерпретация синтаксических деревьев;
— хранение синтаксических деревьев;
— утилитарная поддержка: инкрементальная компиляция и поддержка в IDE.

В обновленной макросистеме макросы исполняются внутри интерпретатора, что позволит обеспечить их переносимость и совместимость. Требование к типизированности синтаксических деревьев основывается на том факте, что среда запуска интерпретатора представляет функцию вывода типов. Более того, интерпретатор синтаксических деревьев *Scala* может потенциально быть использован не только в целях интерпретации макросов, но и, к примеру, для построения в будущем минималистичного REPL.

Проект *Palladium* будет публично представлен на конференции ScalaDays 2014, проходящей с 16 по 18 июня в Берлине.

[1] http://scalameta.org
[2] http://scalamacros.org

3.2.2. Интеграция в IDE

Другим не менее важным применением интерпретатора синтаксических деревьев *Scala* является интеграция данной функциональности в интегрированные среды разработки. Это позволит производить раскрытие макросов средствами IDE, тем самым решая проблему вывода типов *Whitebox* макросов.

Помимо этого, применение интерпретатора позволит упростить управление зависимостями проекта и ускорить инкрементальную компиляцию за счет отсутствия необходимости выполнять многопроходную компиляцию: вначале компиляцию макросов, а затем остальных модулей.

В данный момент проводится работа по реализации конвертера синтаксических деревьев IDE IntelliJ IDEA в деревья, соответствующие новому стандарту синтаксических деревьев *Scala* из проекта *Palladium*.

ГЛАВА 4. ЗАКЛЮЧЕНИЕ

В рамках настоящей работы был разработан интерпретатор типизированных абстрактных синтаксических деревьев языка *Scala*. Интерпретатор поддерживает подавляющее большинство функциональных возможностей языка, за исключением тех, которые не входят в его область применения. Полученное решение позволило разрешить такие проблемы, как вывод типов макросов, устранение многопроходной компиляции при использовании макросов и изоляция среды исполнения макросов.

Предложенный интерпретатор является одним из двух ключевых компонентов новой макросистемы языка *Scala* и входит в проект *Palladium* по реализации обновленной системы макросов языка.

В будущем планируется продолжить работу как над интерпретатором, так и над его инфраструктурой в контексте как макросистемы *Scala*, так и взаимодействия с интегрированными средами разработки.

Одним из последующих направлений по улучшению интерпретатора может быть оптимизация его производительности при помощи JIT-компиляции или, например, построение минимального REPL, базирующегося на интерпретации синтаксических деревьев, а не основанному на механизме интроспекции времени исполнения.

Со стороны интеграции со средами разработки в данный момент проводится работа по реализации конвертера синтаксических деревьев интегрированной среды разработки IntelliJ IDEA в синтаксические деревья, обрабатываемые интерпретатором, для дальнейшей поддержки раскрытия макросов в IDE.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

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

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

3 *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.

4 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.

5 *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 978-1-4503-2064-1. — DOI: 10.1145/2489837.2489840. — URL: http://doi.acm.org/10.1145/2489837.2489840.