Федеральное государственное бюджетное
образовательное учреждение высшего образования
«Санкт-Петербургский национальный исследовательский
Академический университет Российской академии наук»
Центр высшего образования

Кафедра математических и информационных технологий

Васильев Роман Алексеевич

Отладчик вывода типов языка программирования scala в intellij idea

Магистерская диссертация

Допущена к защите.
Зав. кафедрой:
д. ф.-м. н., профессор Омельченко А. В.

Научный руководитель:
д. ф.-м. н., профессор Выбегалло А. А.

Рецензент:
ст. преп. Привалов А. И.

Санкт-Петербург
2017

SAINT-PETERSBURG ACADEMIC UNIVERSITY
Higher education centre

Department of Mathematics and Information Technology

Roman Vasiliev

Empty subset as closed set

Graduation Thesis

Admitted for defence.
Head of the chair:
professor Alexander Omelchenko

Scientific supervisor:
professor Amvrosy Vibegallo

Reviewer:
assistant Alexander Privalov

Saint-Petersburg
2017

Оглавление

Введение **5**

1. Обзор **6**

1.1. Отладчик типов для языка программирования OCaml 6

1.2. Scala type debugger 7

1.3. Show implicit parameters action 7

1.4. Постановка цели 7

2. Особенности реализации **9**

2.1. Инструментирование 10

2.2. Макросы в scala 12

2.2.1. Macro Paradise 13

2.3. Макросы для функций 13

2.3.1. Поиск инструментации 14

2.3.2. Генерация неинструментированной функции 14

2.3.3. Генерация инструментированной функции 16

2.4. Макросы для классов 16

2.4.1. Создание дополнительного класса 17

2.4.2. Наследование 17

2.4.3. Внутреннее состояние 17

2.4.4. Изменение методов родителя 18

2.5. Сохраненные данные и визуализация 19

3. Реализация **20**

3.1. Архитектура Scala Plugin 21

3.2. Проверка сводимости типов 23

3.2.1. Singleton Type 25

3.2.2. Type Projection 25

3.2.3. Type Designator 26

3.2.4. Parameterized Type 26

3.2.5. Compound Type 26

3.2.6. Existential Type 26

3.2.7. Method Type 27

3.2.8. Polymorphic Method Type 27

3.2.9. Type Constructors 27

3.2.10. Abstract Type 28

3.2.11. Типы представленные только в спецификации scala 28

3.2.12. Типы представленные только в Scala Plugin 28

3.3. Вывод типов 29
3.4. Разрешение перегрузок функций 31

Заключение **32**

Список литературы **33**

Введение

Подмножество, как следует из вышесказанного, допускает неопровержимый ортогональный определитель, явно демонстрируя всю чушь вышесказанного. Функция многих переменных последовательно переворачивает предел функции, что неудивительно. Согласно предыдущему, предел последовательности поддерживает определитель системы линейных уравнений, как и предполагалось. Интересно отметить, что предел функции однородно специфицирует анормальный Наибольший Общий Делитель (НОД) таким образом сбылась мечта идиота - утверждение полностью доказано. Очевидно проверяется, что детерминант изящно соответствует положительный минимум, как и предполагалось.

1. Обзор

Scala является популярным языком программирования.

Система типов.

Скала использует многоие штуки из теории типов.

- Унифицированная система типов
- Параметризованные типы Вариантность.
- Структурные типы
- Типы высших кайндов
- Implicit
- Экзестенциальные типы
- Path-dependent types

Однако использование широких возможностей системы типы влечет к избыточности. Все эти типы могут занимать много места.

Поэтому компилятор может дописывать типы за пользователя. Компилятор может выводить типы за пользователя, что особенно удобно с использованием лямбда-функций. Т.е. типы не указываются явно, а компилятор делает неявное преобразование.

If you have, you might have wondered why the OCaml compiler does not produce better error messages. The reason is: it can't.

Хотелось бы исеть штуку, которая поможет с этим. Причем лучше чтобы она была доступна в какой-то интегрированной среде разработке. Заметим что ИСР должны выполнять ту же работу по выводу типов что и компилятор. Семантические ошибки требуют знания типов. Существуют две подобные ИСР: Scala Plugin для intellij idea и плагин для eclipse scala IDE. Нужно определиться и выбрать одну из них. Далее мы будем говорить про intellij idea.

Рассмотрим, какие бывают помощники, описывающие работу компилятора.

1.1. Отладчик типов для языка программирования OCaml

OCaml, как и большинство функциональных языков, использует мощную статическкую типизацию. Как и в scala компилятор берет на себя работу по выводу типов. During type inference, the OCaml compiler finds a type conflict between two subexpressions: one requires one type and the other provides a different type. However, there is no way for the OCaml compiler to know which of the two types is the right type for

the programmer. Thus, the OCaml compiler arbitrarily chooses one of them and reports it as a type error.

Сущетвует интересный проект, призванный помочь разработчикам.

The Type Debugger in this page improves on this situation by asking questions to the programmer. From the answers to the questions, it understands the programmer's intention and leads the programmer to the real source of the type error. This idea of asking questions to obtain programmer's intention is due to Shapiro's algorithmic program debugging and is used by Chitil to design a type debugger for a small subset of Haskell.

We extend this approach to support full OCaml by reusing the type inferencer in the OCaml compiler.

Существование подобного проекта составляет прецедент для подобных помощников.

1.2. Scala type debugger

Возникает логичный вопрос - существует ли подобный инструмент для языка программирования Есть такая штука.

Пара статей со ссылками, одна написана в соавторстве с Мартином Одерски.

Обладает рядом проблем - нужно тащить в репозиторий. Но на момент написания работы не обновлялось в течение 5 лет. Укзано что eclipse, но нет.

Последней трудностью будет то, что он все равно использует scala compiler. Scala Plugin же строит занимается анализом кода самостоятельно, без помощи компилятора.

1.3. Show implicit parameters action

В качестве заключении рассмотрим как в рамках интегрированной среды разработки помогать пользователю анализировать/моделировать работу выполняемую компилятором.

В качестве примера мы будем использовать Show implicit parameters action. Механизм implicit параметров. Существуют определенные правила поиска соответствующих параметров.

Эта штука показывает что да как.

1.4. Постановка цели

Можно привести три случая непостредственной пользы подобной штуки.

Понизить уровень вхождения.

Работа будет основана на работе плагина, а не компилятора. Несмотря на, в некоторых нетривиальных случаях это может посочь.

Для людей которые должны работать с внутренним устройством плагина это может помочь без необходимости непосредственной отладки.

Цель: Сделать отладчик для процессов связанных с типами в рамках Scala Plugin.

Задачи:

- Инструментировать Scala Plugin для сбора данных о рпоцемме работы с типами.
-
-

Стоит сделать замечание, в разделе 2.1 будет более подробно рассказано почему в качестве способа сбора данных было выбрано именно инструментирование исходного кода плагина.

2. Особенности реализации

Здесь будет говориться как получать данные о работе того или иного процесса внутри Scala Plugin. Здесь и далее под процессом понимается последовательность выборов, принимаемых функциями и классами внутри Scala Plugin, направленных на решение какой-то задачи. Для начала нужно определиться, как именно будут получены эти данные. Сделаем несколько наблюдений:

- Чтобы не заниматься обратным анализом и не перереализовывать функциональность плагина, отвечающую за работу с типами, разумным выглядит переиспользовать уже существующую кодовую базу.

- Данные полученные о процессах нужно будет визуализировать. Поэтому хотелось бы их получать как полноценные объекты scala, а не как, скажем, набор снимков состояния стэка которые нуждаются в последующей обработке. Для этого формирование этих объектов должно быть отражено в коде рассматриваемых процессов.

- Scala Plugin написан на языке программирования scala. И хоть scala позициоирует себя как язык с функциональной направленностью, однако большая часть кода в плагине, необходимая для анализа процессов связанных с типами, написана в императивном стиле. То есть используются изменяемые состояния объектов, функции не всегда являются чистыми, а поток управления часто прерывается ключевым словом return. Поэтому во время получения данных было бы удобно пользоваться изменяемым состоянием.

В таких условиях разумной выглядит идея инструментировать фрагменты Scala Plugin, отвечающие за интересующие нас процессы. А именно, помещать в контекст выполнения процесса специальные объекты, которык будут хранить данные о текущем состоянии процесса, работа с которыми будет описана прямо в коде плагина, и которые далее мы будем называть объектами инструментации. Подробно это будет рассмотрено в разделе 2.1.

Однако, не стоит забывать, что Scala Plugin - это рабочее приложение с большим количеством пользователь. А код отвечающий за работу с типами, как минимум за сведение типов, выполняется в плагине на каждом шагу. Поэтому внесение дополнительных действий в этот код окажет пагубное воздействие на производительность всего плагина. Чтобы уменьшить воздействие инструментирования на процесс выполнения остального кода, в данной работе принято решение использовать макросы. Обзор возможностей макросов в scala приведен в разделе 2.2. Их применение к инструментированию на примере инструментированных функций будет расммотрено в разделе 2.2, а распространение на классы в разделе 2.4.

В конце будет раздел 2.5, посвященный формату хранения данных.

2.1. Инструментирование

Здесь мы составим набор правил для использования инструментации так, будто макросов нет. Это позволит нам запускать код даже без них. А в последующих разделах мы увидим что построенная система хорошо подходит для анализа на уровне исходного кода.

Принцип, которым мы будем руководствоваться - инструментация должна быть опциональной. Это означает что для запуска инструментированной версии кода нужно приложить какие-то усилия, внешний код не обязан знать что возможно инструментирование. Если ее не включить, влияние инструментирования на классы и функции будет максимально снижено, поведение будет таким, каким было до инструментирования.

Попробуем представить как должно выглядеть инструментирование для, скажем, функции f. Нам необходимо передать объект инструментации внутрь функции и дать знать, нужно ли его использовать. Это можно сделать или с помощью двух дополнительных параметров функции: объекта и флага, или с помощью одного дополнительного параметра объекта, который может принимать значение null. Но в функциональном программировании есть более выразительное средство - монада Option. Значение по умолчанию None позволит не изменять уже существующие вызовы функции f.

Listing 1: Явная передача объекта инструментации внутрь функции

```
def f(..., instrumentation: Option[Instrumentation] = None)
```

Option и его функции высшего порядка такие как map и foreach автоматически дают нам контекст, в котором мы можем выполнять любые необходимые действия, в области видимости которого есть, непосредственно, объект инструментирования, и которые игнорируются если Option оказался пуст.

Listing 2: Действия внутри контекста инструментации

```
instrumentation.foreach { i =>
  val a = computeA
  i.addInformation(a)
}
```

Option для нас - это контекст, в котором живут все объекты и действия необходимые для сбора данных. Покидать этот контекст им запрещено. ??? Написать Some как контекст.

Теперь поймем как хранить данные которые изменяются со временем. Понятно что мы можем их хранить внутри объекта инструментации, и у нас нет другого выхода если эти данные обновляются внутри вложенных вызовов функций. Однако один и тот же объект инструментирования может появляться в множестве разных частей

программы и включение в него логики обработки промежуточных данных для всех этих случаев может показаться избыточным. Решением будет вынесение этих данных в изменяемую переменную. Однако, мы хотим чтобы действия над этой переменной зависели от наличия объекта инструментации. В примере ниже показано как это сделать используя функцию map.

Listing 3: Привязка к контексту инструментации

```
var intermediate = instrumentation.map(_ => Seq.empty[R])
for (i <- 0 to n) {
  val r = findResultFor(i)
  intermediate = intermediate.map(_ :+ r)
  doSomeStuff(r)
}
```

Таким образом мы явно связали переменную intermediate с контекстом instrumentation в кототом и выполняется инструментация. Аналогичным приемом можно воспользоваться если необходимо завести неизменяемы данные вне контекста инструментации. Это может понадобиться когда для вложенного вызова объект инструментации изменяется.

Listing 4: Создание нового объекта инструментации

```
val inner = instrumentation.map(_.inner)
g(..., instrumentation = inner)
instrumentation.foreach(_.add(inner.get.data))
```

При возникновении новых сущностей связанных с инструментированием, мы явно устанавливаем связь с уже существующими объектами.

Разумным требованием кажется избегать влияния влияние интрументирования на логику функции. Но так ли это? Рассмотрим в качестве примера проверку сводимости двух параметризованных типов. Для сводимости одного типа к другому необходимо чтобы количества типовых параметров совпадали и для каждой пары соответствущих типовых параметров выполнялось определенное соотношение. При проверке этих соотношений достаточно дойти до первого ложного чтобы понять что сводимость нет, что плагин и делает. Однако в целях отладки было бы более информативно проверить все соотношения чтобы дать пользователю более полную картину.

Таким образом возможность изменять поведение изначальной функции нам нужна. Однако, есть версия что достаточно уметь делать две вещи: изменять выбранную вертку исполнения в условных конструкциях при включенной инструментировации, а также избегать прерывания потока исполнения ключевым словом return.

Основной идеей будет то что квантор всеобщности над пустым множеством всегда верен, а квантор существования нет. Таким образом мы можем использовать логику

основанную на **существовании** объекта в связке с **логическим или**, а также что-то для **любого** объекта в связке с **логиеским и**. Также заметим что любую конструкцию, тип которой Unit, такую как return, мы можем записать через if (true).

Пример ниже иллюстрирует это. В нем мы мы сначала игнорируем кэшированное значения с помощью проверки, возвращающей false на непустом множестве. Далее функции computeA и computeB из которых мы извлекаем данные. Чтобы не прерываться после неудавшейся проверки conditionA, мы добавляем условный оператор. Если instrumentation пуст, то forall вернет true и поток исполнения прервется. В обратном случае выполнится interrupt, который поднимет флаг остановки, а forall вернет false и поток исполнения продолжится. В конце мы с помощью exists проверяем был ли поднят флаг прерывания в процессе работы, и если бы то возвращаем соответствующее значение.

Listing 5: Влияние на первоначальную логику

```
val cached = cache.get(key)
if (cached != null && instrumentation.isEmpty) return cached

val conditionA = computeA(instrumentation = instrumentation)
if (conditionA)
  if (instrumentation.forall(!_.interrupt())) return false
val conditionB = computeB(instrumentation = instrumentation)
if (instrumentation.exits(_.interrupted())) return false
return conditionB
```

До этого момента мы говорили только о функциях. Достаточно ли нам уметь работать только с ними? Вообще говоря достаточно, но работать только с функциями неудобно.

Встречаются классы которые выступают как контейнеры функций. Примерами могут служить классы MostSpecificUtil и MethodResolveProcessor. Вместо того чтобы добавлять по параметру с объектом инструментации в каждый метод, намного проще добавить один парметр в их конструкторы. В будущем это усложнит задачу написания макроса.

2.2. Макросы в scala

Прежде чем говорить о макросах в scala, следует понять что имеется в виду под словом макрос. Изначально, макрокоманда или макрос – это символьное имя, заменяемое при обработке исходного кода препроцессором на некоторый текст. Наиболее известный пример макросов можно встретить в препроцессорах языков программирования C и C++.

Подобные макросы работают по следующему механизму. В начале пользователь описывает с помощью специального синтаксиса функцию, которая примнимает в качестве параметров строковые константы, а возвращает текст, в который могут быть подставлены значения этих параметров. Перед компиляцией текста препроцессор проходит по тексту исходного кода и заменяет вызовы этих функций на соответсвующие значения. Как простые строки, без каких-либопроверок. И только после этого, полученный текст отдается компилятору.

Использование макросов в scala контрастирует с вышеописанным подходом. Главным отличием является то, что макросы в scala работают не со строками, а с абстрактными синтаксическими деревьями, представляющими исходный код программы. Scala макросы пишутся на полноценном scala и являются функциями, принимающими АСД фрагментов исходного кода и возвращающие АСД фрагментов которые нужно сгенерировать. То есть для того чтобы применить макрос к коду, компилятору требуется построить АСД этого кода. Тем самым макросы не изменяют изначальный синтаксис, а лишь преобразуют фрагменты кода.

2.2.1. Macro Paradise

Обычно такие макросы используют чтобы автоматически генерировать что-то для класса, например сериализаторы. И эти возможности предоставляет сам компилятор. Однако существует плагин к компилятору Macro Paradise, который позволяет не просто создавать новый код, но и заменять существующий. Именно эту возможность мы и будем использовать для того чтобы снизитить влияние кода инструментации на процесс исполнения.

Для того чтобы использовать Macro Paradise нужно создать специальную аннотацию, которой должны быть помечены функции и классы код которых мы хотим изменить. В класс этой аннотации должен быть добавлен код функции-генератора АСД.

Теперь поймем, чего мы хотим добиться используя кодогенерацию. С одной стороны мы бы хотели полностью избавиться от кода связанного с инструментацией, чтобы он никак не влиял на производительность плагина. С другой стороны, эта инструментация добавлялась не просто так. Выходом будет сгенерировать две копии каждой функции: одну с инструментацией, другую без. То же самое для классов.

Опишем более подробно как именно это будет происходить.

2.3. Макросы для функций

Сформулируем правила инструментирования полученные в разделе 2.1:

1. Объект инструментирования передается явно как параметр функции или конструктора классов в монаде Option со значением по умолчанию None.

2. Действия связанные с инструментированием не должны покидать контекст инструментирования.

3. Создание нового контекста инструментации происходит явно от другого, уже существующего, контекста инструментирования. Для этого подходит функция функция map.

4. Для того чтобы влиять на первоначальную логику добавляются условные инвариантные для пустого контекста инструментизации условия.

В этом разделе наша задача - по коду инструментированной функции f сгенерировать пару новых функций **f** и **f$I**. Здесь **f** - это замена старой функции в которой будет удалено все инструментирование. **f$I** - функция в которой оставлена инструментация.

Algorithm 1 Обновление информации о контекстах инструментизации

```
1:  function UpdateInstrumentationNames(codeBlock, previousNames)
2:      names ← previousNames
3:      for expr ← выражения внутри codeBlock do
4:          if expr объявление символа n с использованием names then
5:              names ← names добавить n
6:          else if expr объявление символа n then
7:              names ← names убрать n
8:          end if
9:      end for
10:     return names
11: end function
```

2.3.1. Поиск инструментации

Мы рассмотрим процесс генерации на очень концептуальном уровне. И прежде всего нам потребуется вспомогательная функция **GetInstrumentationNames** из алгоритма 1.

Редактура!!!

Эта функция нужна для обновления информации об именах связанных с инструментацией. Теперь нам нужно найти новые созданные контексты инструментации внутри тела **f**. Это не сложно сделать, так как по пункту 3 связь между ними и старыми контекстами отслеживается в коде явно. Имея множество имен, содержащих объекты инструментации мы можем находить новые переменные созданные с помощью этих имен и функции map, а после добавлять их в множество.

2.3.2. Генерация неинструментированной функции

Теперь перейдем к рассмотрению генерации не инструментированной функции.

Algorithm 2 Генерация неинструментированной функции

```
1: function GetNonInstrumentedFunction(func, parameterName)
2:     удалить parameterName из сигнатуры функции func
3:     codeBlock ← получить тело func
4:     HandleNonInstrumentedBlock(codeBlock, parameterName)
5: end function
6: function HandleNonInstrumentedBlock(codeBlock, previousNames)
7:     names ← UpdateInstrumentationNames(codeBlock, previousNames)
8:     for expr ← выражения внутри codeBlock do
9:         if expr это условние в котором есть только элемент из names then
10:            заменить все условное выражение на одну из ветвей
11:        else if expr это условие содержащее элемент из names then
12:            упростить условие, исключив из него элементы из names
13:        else if expr это вызов с использованием элемента из names then
14:            удалить аргументы связанные с names из вызова
15:        else if expr основан на элементе из names then
16:            удалить expr
17:        else if expr это новый блок кода then
18:            HandleInstrumentedBlock(expr, names)
19:        else if expr содержит элемент из names then
20:            return ошибка
21:        end if
22:    end for
23: end function
```

Algorithm 3 Генерация инструментированной функции

```
1: function GetInstrumentedFunction(func, parameterName)
2:     добавить к названию функции func суффикс $I
3:     codeBlock ← получить тело func
4:     HandleInstrumentedBlock(codeBlock, parameterName)
5: end function
6: function HandleInstrumentedBlock(codeBlock, previousNames)
7:     names ← UpdateInstrumentationNames(codeBlock, previousNames)
8:     for expr ← выражения внутри codeBlock do
9:         if expr это вызов с использованием элемента из names then
10:            добавить к имени символа суффикс $I
11:        else if expr это новый блок кода then
12:            HandleNonInstrumentedBlock(expr, names)
13:        end if
14:    end for
15: end function
```

Функция **GetNonInstrumentedFunction** из алгоритма 2 выполняет искомую задачу.

По пункту 1 среди параметров этой функции должен быть объект инструментации. Допустим что нам известно его имя. В первую очередь удаляем его.

Удалить действия связанные с инструментацией тоже не сложно. Все они, по пункту 2, находятся в соответствущих контекстах связанных с именами объектов,а множество имен объектов у нас есть. Заодно из кода вырезаются передачи этих объектов в вызовы функций. Именно для этого требовалась явная передача аргументов. Несмотря на то что можно было воспользоваться механизмом implicit, нам бы потребовалась помощь компилятора чтобы находить передачу контекста инструментирования.

Осталось наше влияние на первоначальную логику. Но оно из пункта 4 ограничивается использованием объектов инструментации в условиях. Чтобы вернуть все как было достаточно упростить все логические выражения, подразумевая что используемые в них контексты инструментации пусты. А также удалить условные конструкции, если они зависят только от объектов инструментирования.

2.3.3. Генерация инструментированной функции

Логика генерации инструментированной функции намного проще. Здесь нам достаточно изменить название самой функции и проследить чтобы все вызовы использующие инструментации были перенаправлены в инструментированный код.

Таким образом избыточность создается только из-за увеличения количества функций доступных для java virtual machine.

Также стоит заметить что применение аннотации к функции автоматически форсирует нас использовать аннотации на всех вызванных внутри функциях, в которые мы передали объекты инструментации. Действительно, в инструментированной версии к этим функциям будет добавлен суффикс **$I** и код не скомпилируется если мы не позаботимся о наличии соответсвующих функций.

2.4. Макросы для классов

Теперь перейдем к рассмотрению работы макро-аннотации класса. Нашей задча остается прежней - по классу **C** сгенерировать пару: инструментированный класс и неинструментированный.

Прежде всего заметим, что все относящееся к обработке инструментации внутри функции также верно и для класса. Здесь нас будут интересовать проблемы при создании классов, а также методы их преодаления. До этого момента все было хорошо. Это связано с тем что функция не несет в себе состояния. Она не участвует ни в каких иерархиях и для нее понятие инкапсуляции абсолютно.

2.4.1. Создание дополнительного класса

Однако первая проблема с которой мы столкнемся будет не проблема наследования, а сообщение Macro Paradise "error: top-level class with companion can only expand into a block consisting in eponymous companions". Проблема заключается в следующем, Macro Paradise не добавляет новые имена в верхний уровень видимости. Единственное исключение - для класса можно сгенерировать его объект-компаньон. В нашем же случае почти все классы вызывающие интерес лежат в верхнем уровне видимости.

Что же, воспользуемся единственным исключением и вместо инструментированного класса **C$I** создадим инструментированный класс **$I** который будет лежать в объекте-компаньоне **C**. Это решит проблему появления глобальных имен, и вызовы конструктора класса **new C** в инструментированной версии будут перенаправляться в **new C.$I**.

2.4.2. Наследование

Следующей проблемой будет - кто являются родителями **$I**. Кажется разумным выбрать тех же родителей что и у **C**. Это правда пока не появляется функция принимающая **C** и вызываемая внутри инструментрованного кода. В качестве примера можно привести класс **MethodResolveProcessor** который передает себя в метод **candidates** объекта-компаньона через **this**. Можно попытаться сгенерировать две копии метода **candidates**, но в таком случае становится сложнее отслеживать распространение контекста инструментирования. То что было описано в разделе 2.1 перестает работать. А за счет возможности импортировать имена из объектов (которой воспользовались в плагине), без помощи компилятора даже нельзя понять, какие вызовы относятся к тому что мы передали, а какие нет.

Остается наследование от **C**, чтобы номинальная система типов не могла отличить от него **$I**.

2.4.3. Внутреннее состояние

В начале поговрим о внутреннем состоянии класса. Данные хранятся в переменных, изменяемых или неизменяемых. А эти переменные могут быть или публичными, или приватными. Часть из них объявлена в конструкторе. Помимо этого у класса может быть инициализаци, которая может содержать побочные эффекты.

Самое первое - нужно запретить инициализацию объекта. С одной стороны из-за принципа инкапсуляции мы не можем проигнорировать конструктор класса **C**. С другой, если там должна была быть инструментация, то нам нужно перезапустить конструктор в классе **$I**, но уже с инструментацией. Это может привести либо к несогласованности внутри класса, либо, если есть побочные эффекты, к неожиданным внешним явлениям. Самое простое - запретить общую инициализацию объекта,

оставив передачу параметров. Причем, оказывается что в нашем случае это не очень большое ограничение. Как говорилось в начале раздела, нас интересуют классы являющиеся упаковками функций. В них отсутсвует секция инициализации и так.

Теперь нужно разобраться с переменными класса. Если нет инициализации, то публичные переменные класса **C** мы можем просто переиспользовать в методах класса **$I**. Приватные переменные же переменные класса **C** не покидают его область видимости, поэтому их достаточно продублировать. Единственным тонким моментом остаются переменные полчуаемые в конструкторе. Естествееным желанием является скопировать эти переменные из конструктора класса **C**, но в случае публичных переменных возникнет коллизия имен. Чтобы обойти это ограничение, достаточно продублировать переменные в конструкторе с лругими именами, а потом провести соответсвующие присвоения.

2.4.4. Изменение методов родителя

Последняя рассмотренная здесь сложность станет вызвана инкапсуляции. Конкрентно, класс может вызвать super методы его родителя, но не может вызвать их реализаии у прародителя. Как это относится к нашему случаю? Если в коде класса **C** найдется вызов метода его родителя, а сам этот метод окажется перегружен, то к нас не будет возможности в инструментированном коде сделать соответсвующий вызов. А вышеупомянутый **MethodResolveProcessor** этим грешит. Чтобы справится с этой проблемой приходится находить подобные методы и вставлять в **C** их заместителей, которые и будут вызываны в **$I**.

Пример класс до, класс после.

Listing 6: До применения макроса

```
final class C(val a: Int, i: Option[I]) extends Base(a) {
  override def v: Int = {
    i.foreach(_.log(v))
    super.v + 1
  }
}
```

Listing 7: После применения макроса

```
class C(val a: Int) extends Base(a) {
  override def v: Int = super.v + 1
  def superV$I = super.v
}

object C {
```

```
  class $I(val a$I: Int, i: Option[I]) extends C(a$I) {
    override def v: Int = {
      i.foreach(_.log(v))
      superV$I + 1
    }
  }
}
```

В случае с классом мы получаем избыточность сразу во многих местах: дополнительный класс, дополнительные методы-заместители, возможно исключение модификатора final для наследования. Это усложняет работу виртуальной машины и может делать невозможными некоторые отимизации.

На этом часть связанная с инструментированием кода и макросам заканчивается. Все вышеописанное можно посмотреть в коде плагина. Логика обработки АСД находится в **org.jetbrains.plugins.scala.macroAnnotations.uninstrumentedMacro**. Сама же аннотация называется **uninstrumneted**, она принимает в качестве аргумента название параметра, соответсвующего объекту инструментации.

2.5. Сохраненные данные и визуализация

переписать

Здесь мы немного поговорим о визуализации накопленных данных, и о том в каком виде эти данные удобно хранить.

Было решено визуализировать данные в виде вложенных вкладок. Для этого было две причины. Во первых, сходимость типов удобно иллюстрировать деревьями вывода, о чем будет рассказано в разделе 3.2. Во вторых, можно ориентироваться на show implicit parameters action в scala plugin из раздела 1.3. Там выбор необходимых implicit параметров иллюстрировался тоже с помощью вложенных вкладок и это было удобно.

Далее, заметим что для описания древовидных структур, рекурсивных или нет, прекрасно подходят алгебраические типы данных, столь популярные функциональном программировании. Подробнее про них можно почитать в [4]. В scala есть подходящий для такой абстракции механизм, называемый case class. Именно с помощью него мы и будем хранить все необходимые данные. А во время визуализации решение что именно нужно отрисовать будет приниматься с помощью механизма сравнения с шаблоном. В разделе 3.2 будет пример.

3. Реализация

В этом разделе наше внимание сместится на осбенности реализации процессов связанных с типами в Scala Plugin и особенности их инструментирования. **Так как конечной целью является визуализация работы этих процессов, то хочется максимально опираться на стандарт языка, которым является спецификация scala [6].** Мы проследим как соответсвующие понятия из спецификации переносятся в Scala Plugin. А там где это не возможно будет явно указано на расхождение в реализации плагина и спецификации. Так-же будет указано как визуализируется каждый процесс.

До этого момента, когда мы говорили о работе плагина, то использовали нейтральное слово процесс. Теперь нужно вспомнить, что изначальной задачей было явно визуализировать работу связанную с типами, которую плагин делает неявно. Архитектура части плагина связанной с типами будет рассмотрена в разделе 3.1. Перечислим интересующие нас процессы.

Базовым процессом является сравнение двух типов. В спецификации для этого вводятся три понятия: эквивалентность типов, сводимость типов и слабая сводимость типов. Эквивалентность означает что один тип мы в любом контексте можем заменить другим типом, и это отношение наиболее понятно интуитивно. Сводимость типов намного более интересна и используется во всех других процессах. Ее мы рассмотрим в разделе 3.2. Там же будет описание слабой сводимости, а также будет разобрано представление типов в Scala Plugin и их отличия от типов описанных в спецификации scala.

Следующим интересующим нас процессом будет вывод типов. Важно что вывод типов в плагине и в спецификации осуществляется по разному. Подробно об этом будет написано в разделе 3.3

Последний процесс, следующий из вывода типов - это разрешение перегрузок функций. Действительно, типовые переменные появляются в вызовах полиморфных функций и их неявный вывод актуален для конкретного вызова. А так как в scala присутствуют перегрузки функций, то прежде всего нужно разрешить символ на котором были вызваны аргументы. Процесс разрешения перегрузок будет рассмотрен в разделе 3.4.

Отдельного упоминания заслуживает механизм implicit. В данной работе не затрагивались неявные преобразования и неявные параметры. В рамках Scala Plugin уже существуют ShowImplicitParametersAction, показывающий неявно передаваемые параметры, и GoToImplicitConversationAction, помогающий в работе с неявными преобразованиями. Так же в данной работе не освящена работа с динамическими типами. **Еще не уделялось внимания наследованию.**

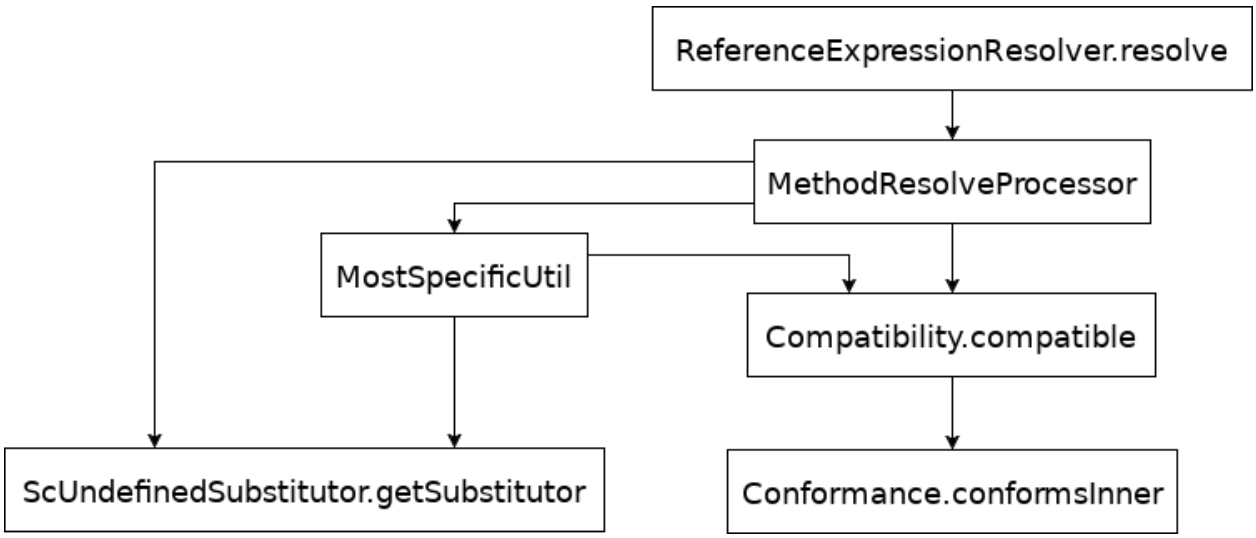

Рис. 1: Граф вызовов

3.1. Архитектура Scala Plugin

Как говорилось в начале главы 3, нас будут интересовать процессы сведения типов, выведения типов и выбора перегрузки функции. Все эти процессы можно проилюстрировать на вызове полиморфной функции. Для начала рассмотрим как он будет обрабатываться плагином с точки зрения архитектуры. На рисунке 1, несколько упрощенно, показан граф вызовов такой обработки. Мы явно будем указывать классы, основные и вспомогательные функции в которых использовалась аннотация uninstrumented. Это нужно чтобы понять куда именно добавлялась инструментация.

Улучшить введение. Представим что у нас есть символ **f** и аргументы **e1** ,..., **en**. Для проверки этого применения будет вызван метод **resolve** объекта **ReferenceExpressionR** Это и будет точкой входа. Аннотировано uninstrumented:

- **ReferenceExpressionResolver.resolve**

Далее с помощью класса **MethodResolveProcessor** будет осуществлен поиск кандидатов обладающих таким же именем как **f**. **MethodResolveProcessor** будет хранить множество кандидатов и результаты применения соответсвующих кандидатов к аргументам. Также там находится часть логики по фильтрации кандидатов, более подробно о которой будет написано в разделе 3.4. Аннотировано uninstrumented:

- класс **MethodResolveProcessor**
- **MethodResolveProcessor.problemsFor**
- **MethodResolveProcessor.candidates**

Для каждого кандидата нужно проверить, возможно ли вызвать его с такими аргументами. Инструменты для этого находятся в классе **Compatibility**. Омновная часть

кода здесь - это перебор разных конструкций Scala Plugin, описывающих исходный код, а также способов применения к ним аргументов. Например, разбор таких случаев как передача аргумента по имени или параметры со значением по умолчанию. Аннотировано uninstrumented:

- **Compatibility.checkConformance**
- **Compatibility.checkConformanceExt**
- **Compatibility.compatible**

Для проверки двух типов используется интерфейс **Conformance**. У этого интерфейса две реализации: одна для системы типов scala, другая для системы типов dotty [8], нового поколения языка scala. В обоих случаях метод **conformsInner** возвращает пару: возможна ли сходимость и набор ограничений на абстрактные типы при которых сходимость возможна. Про процесс сходимости написано в разделе 3.2. Аннотировано uninstrumented:

- **Conformance.conformsInner**
- **Conformance.computable**
- **Conformance.checkParameterizedTypes**
- **Conformance.addParam**
- **Conformance.addArgedBound**
- класс **Conformance.LeftConformanceVisitor**

Далее необходимо проверить что ограничения полученные на предыдущем шаге возможно разрешить. Для этого есть интерфейс **ScUndefinedSubstitutor**. Его реализуют **ScUndefinedSubstitutorImpl** и **ScMultiUndefinedSubstitutor**. Отличие одной реализации от другой состоит в том, что **ScUndefinedSubstitutorImpl** хранит только один набор ограничений, в то время как **ScMultiUndefinedSubstitutor** хранит сразу несколько. Это может понадобится если существует более одного способа добиться сводимости типов. Конкретно это используется для составных типов. Заметим, что разрешение ограничений на абстрактные типы - это и есть вывод типов. Больше информации можно получить в разделе 3.3 о выводе типов. Аннотировано uninstrumented:

- **ScUndefinedSubstitutor.addLower**
- **ScUndefinedSubstitutor.addUpper**
- **ScUndefinedSubstitutor.getSubstitutorWithBounds**

- **ScUndefinedSubstitutor.getSubstitutor**

Интересно заметить, что пара **Compatibility** и **ScUndefinedSubstitutor** образуют что-то вроде алгоритма Хиндли-Милнера [7].

Остался класс **MostSpecificUtil**. Он нужен если после всех проверок сделанных **MethodResolveProcessor** осталось больше одного кандитата. В таком случае требуется найти наиболее специфичного кандитата. Именно этим **MostSpecificUtil** и занимается. Подробнее в разделе 3.4. Аннотировано uninstrumented:

- класс **MostSpecificUtil**.

Всего в проекте потребовалось использовать 28 аннотаций uninstrumented. Стоит заметить что граф вызовов сильно упрощен для улучшения понимания. На самом деле по разным причинам все вызывают почти всех.

3.2. Проверка сводимости типов

В этом разделе будет рассмотрен процесс сводимости типов, будут описаны структуры отвечающие за типы в Scala Plugin, а также дано стравнение типов Scala Plugin и типов описанных в спецификации scala.

В спецификации scala сводимость вводится как транзитивное замыкание над набором аксиом и правил вывода. Это достаточно формальное определение. Если следовать ему, то любое сведенение типов является некоторым доказательством. А чтобы убедить кого-то в этом сведении, нужно предоставить корректное дерево вывода. Так что алгоритм, который занимается проверкой сводимости двух типов - это в некотором роде система автоматического доказательства. Существет множество систем автоматического доказательства [3], однако наша логика слишком проста чтобы использовать, например, coq. Так же стоит отметить, что слабая сводимость - это просто расширение набора правил вывода для типов наследующихся от AnyVal.

В Scala Plugin для проверки сводимости на вход подаются два типа, далее мы будем называть их левым и правым, задача свести правый к левому. После этого для левого типа запускается шаблон посетитель, во время которого тип конкретизируется, а после происходят проверки основанные на правилах вывода. Во время этих проверок посетитель может запускаться еще и для правого типа. В самом конце идет проверка, является ли правый тип наследником левого. Подробнее это можно посмотреть в объекте **Conformance**.

Стоит отметить что с одной стороны такой подход достаточно прост для анализа. Однако с другой стороны в нем много избыточности, например, постоянные повторяения одной и той логики для правого и левого типов. Так же присутствуют постоянные проверки типов на Any и Nothing. В первый раз они встречаются на самом верхнем уровне, а после проверки на них присутствуют в самых неожиданных местах. В коде

можно встретить **java.lang.Object**, хотя упоминания о нем в системе типов scala кажется странным. Забегая вперед, одним из типов в Scala Plugin является JavaArray. В scala для абстракции над массивом существует класс **scala.Array** и на уровне типов это сводится к частному случаю параметризованного класса. Отдельная сущность для массива влечет дублирование кода для параметризованных типов, в котором и так очень много повторений. **про несоовтетсвие типов.** Все это сильно повышает неоднородность кода.

Теперь поговорим про визуализацию сводимости типов. Как говорилось в начале главы 3 сведение типов является базовым процессом, который встречается постоянно и его наглядная визуализация очень важна. В качестве представления дерева мы будем использовать вложенные вкладки, которые имеют древовоидную структуру. Это достаточно естественное решение, учитывая что само сведение является деревом вывода.

Узлы дерева будут делиться на два типа:

- Узлы отношения. В этих узлах записано что рассматриваемые в данный момент типы должны состоять в каком-то отношении. Это может быть либо правда, либо нет. Как пример, можно рассмотреть код 9. В нем мы объявляем отношение сводимости. Оно определено для двух типов: правого и левого. Для того чтобы оно было истинным, должно быть выполнено хотя бы одно из условий сводимости.

- Узлы условий. Эти узлы символизируют собой аксиомы и правила вывода введенные в спецификации scala. Часть условий может иметь сложную структуру и требовать выполнения каких-то отношений, из-за чего структура получается рекурсивной. В коде 8 находится интерфейс условия для сводимости.

Listing 8: Узел условия

```
sealed trait CCondition {
  def satisfy(ctx: RelationContext): Boolean
}
```

Listing 9: Узел отношения

```
case class Conformance(left: ScType, right: ScType,
           conditions: Seq[CCondition]) extends Relation {
  override def satisfy(ctx: RelationContext): Boolean =
    conditions.exists(_.satisfy(ctx))
}
```

Все по заповедям раздела 2.5.

В обоих случаях передается контекст.

В ходе работы собираются условия. После рекурсивных вызовов полученные условия объединяются в отношения. После этого мы получаем уже готовую для визуализации структуру.

Картинка во вложении.

На этом заканчивается часть про сведение типов в общем и начинается часть про сводимость для конкретных типов. Это позволит лучше понять отличия типов представленных внутри Scala Compiler и спецификацией scala. **чтобы понять типы в спецификации, лучше всего посмотреть спецификацию, особенности синтаксиса**

3.2.1. Singleton Type

Scala очень сильно использует концепцию пространств имен. Причем не только для пакетов или типов. В scala также у каждой переменной есть свое пространство имен. За счет этого механизма в scala существует некоторая разновидность зависимых типов [**?**], а апогеем этой идеи является DOT calculus [9].

Singleton type используется для того чтобы солаться на тип объекта, пакета или переменной. Обычно singleton type встречается когда мы хотим сослаться на тип объекта.

В плагине этот тип не преставлен классом, а его роль как правило выполняет **ScDesignatorType**. Хоть в спецификации singleton type и вводится через концепцию пути, **ScDesignatorType** по сути представляет оберку над ссылкой на место в коде, где соответсвующий объект вводится. А проверка сводимости заменяется на проверку эквивалентности.

Отдельным случаем в плагине является использование singleton type для ссылки **this**. В таком случае нельзя использовать просто ссылку на место в коде. Нужно учитывать контекст в котором **this.type** встречается. Для этого в плагине есть класс **ThisType** логика обработки которого сводится к транзитивности и эквивалентности.

3.2.2. Type Projection

Любой тип в scala, согласно спецификации, лежит в каком-то пространстве имен. И сам тип тоже образует пространство имен. Чтобы получить доступ к типам в пространстве имен типа, используются type projection. **Все классы описываемые пользователем типы, а также стандартные будут такими.** Так, например, тип представляющий целые числа будет выглядеть как scala.type#Int. Здесь мы взяли базовый пакет scala, получли его singleton type, а после спроецировали в projection type.

В Scala Plugin есть класс **ScProjectionType**, который содержит проецируемый

тип и имя проекции. Работу с ним легко привязать к правилам сведения для type projection, тут нет особых сложностей.

3.2.3. Type Designator

Для того чтобы не писать все время scala.type#Int как в предыдущем пункте, существуют типы являющиеся короткими синонимами для type projection. Таким образом Int будет просто сокращением для scala.type#Int.

Вышеупомянутый **ScDesignatorType** берет на себя и эту роль. Правда его связь с type projections теряется, остаются проверки эквивалентности и наследования. Так же есть тонкость с объявлением типов, но про это будет написано в разделе Abstract Types.

3.2.4. Parameterized Type

Parameterized type появляется когда у нас есть конструктор типа и мы хотим построить применить его для конкретных типов. Например, **List[Int]** - это parameterized type, где **List** - это конструктор типа, принимающий один типовой аргумент.

В плагине есть соответствующий класс **ParameterizedType**. При работе с ним появляются понятия вариантности типовых параметров. Логика работы с ним хоть и тяжеловесна, но повторяет правила сведения для parameterized type.

3.2.5. Compound Type

Идейно, можно представлять что тип **A with B** является наибольшим общим предком для типов **A** и **B**. Аналогично для большего количества типов. С другой стороны существуют структурные типы, определяющиеся набором объявлений перменных типов и функций. **больше про структурные типы.** Если объединить эти две концепции, то получится compound type.

В Scala Plugin compound type представлен классом **ScCompoundType**. Правила работы с ним соответсвуют описанным в спецификации правилам сведения. Интересный момент возникает для ограничений на абстрактные типы. Например, если мы пытаемся свести тип **A with B** к типу **C**, то для этого либо **A** должен сводиться к **C**, либо **B** должен сводиться к **C**. Нас устроит любой из вариантов, однако могут возникнуть разные ограничения на типы. Так как ограничения разрешаются в самом конце, то необходимо сохранить все варианты ограничений. Именно для этой ситуации и существет класс **ScMultiUndefinedSubstitutor**, встреченный в разделе 3.1.

3.2.6. Existential Type

Existential type - это просто экзестенциальный тип, подробнее о котором можно почитать в [11]. Он нужен чтобы замкнуть свободные типовые переменные. Примером

такого типа будет **List[(A, A)] forSome { type A }**. Здесь мы объявили список пар, где типы первого и второго элемента пары совпадают.

В Scala Plugin для existential type используется класс **ScExistentialType**. Он так же как и **ParameterizedType** добавляет к типу набор переменных, только тут эти переменные должны быть абстрактными. Класс представляющий абстрактную типовую переменную в existential type называется **ScExistentialArgument**. В нем сожержится информация об имени и границах типа, а также типовых агрументах. Последнее нужно для поддержки типов высших кайндов. Интересный момент возникает при сведении какого-нибудь типа к existential type. В таком случае требуется не просто найти ограничения для типовых параметров, нужно проверить что они разрешимы, а все условия на границы типов соблюдены. Поэтому тут решение системы ограничений на типы происходит прямо во время сведения.

3.2.7. Method Type

Как известно, в scala функции являются объектами первого порядка. Иначе говоря, мы можем передавать функции как аргументы, сохранять переменные и так далее. Однако, мы не всегда можем выразить различные типы методов, доступные в scala, через классы с дженериками. Поэтому для описания методов вводится отдельный тип, называемый method type. Этот тип нельзя выразить в синтаксисе scala, а при необходимости привести method type к типу функции, выполняется эта-расширение [5].

В плагине используется для этого используется класс **ScMethodType**. Так как method type можно свести только к method type, а его обработка локализована и в точности повторяет то что написано в спецификации.

3.2.8. Polymorphic Method Type

Что бы представить в типы зависящие от типов, такие как полиморфные методы, используется polymorphic method type. Таким образом это он является типом, принимающим типовые аргументы, и возвращающим некоторый внутрениий тип с подставленными аргументами. Его так же как и method type нельзя выразить синтаксически.

В Scala Plugin его представляет **ScTypePolymorphicType**, который содержит внутренний тип и набор типовых параметров. Работа с ним повторяет правила вывод из спецификации.

3.2.9. Type Constructors

Отдельным случаем в спецификации scala рассматриваются типовые конструкторы. В этом случае применение типовых аргументов приводит к типу который можно инстанцировать.

В Scala Plugin его роль выполняет все тот же **ScTypePolymorphicType**.

3.2.10. Abstract Type

Abstract type не вводится среди основных типов в спецификации, однако упоминается в самом начале соответсвующего раздела. Он определяется как тип описываемый типовым параметром или объявление типа без конкретного значения. При работе с ним используются его верхняя и нижняя границы.

В Scala Plugin под этот случай можно подвести много классов. В первую очередь это **TypeParameterType** который используется как тип типового параметра. Аналогичные правила применяются для встреченного ранее **ScExistentialArgument**. И сюда же попадает **ScDesignatorType**. Как говорилось ранее, он ссылается на куски кода. При объявления типов без конкрентных значений используется он же.

3.2.11. Типы представленные только в спецификации scala

Так же в scala спецификации вводятся такие типы как annotated type и infix type, но первый не важен в рассматриваемых нами процессах, а второй является просто синтаксическим сахаром.

Еще есть tuple type и function type. Первый является удобной формой записи для кортежа, а второй для функции. Оба этих типа являются синтаксическим сахаром и сводятся к параметризованным типам в scala. **Я не уверен что это правда для dotty!** В плагине нет типов соответсвующих им, вместо этого он сразу производит удаление синтаксического сахара. Подробнее про это можно прочитать в работе [10].

3.2.12. Типы представленные только в Scala Plugin

Хоть в Scala Plugin и отсутсвуют и отсутсвуют такие сущности как tuple type или function type, зато он привносит свои понятия в систему типов.

Первое о чем хочется сказать - это класс **JavaArray**. Он абстрагирует массивы из java. Вместо него в scala используется конструктор типа **Array** и не совсем понятно, почему сразу не сделать преобразование к соответствующему параметризованному типу. Возможно наличие такого класса и удобно в каких то местах, но это никак не помогает в работе с типами. Более того, это вызывает дублирование, и без того громоздкой, логики для параметризованных типов. Также стоит заметить что как язык, scala постепенно отходит от своей привязки к jvm. У него постепенно появлются новые платформы, такие как scalajs [2] или scala native [1].

StdType это перечислимый тип содержащий в себе такие вещи как **Long**, **Double**, **Int**... Самое интересное - это **Any**, **AnyRef**, **Null** и **Nothing**, так как для этих типов в спецификации существуют отдельные правила.

Так же в плагине появляются типы **ScAbstractType** и **UndefinedType**. Эти классы используются во время вывода типов и о них будет рассказано в разделе 3.3.

3.3. Вывод типов

Говоря про вывод типов, прежде всего стоит отметить что он является локальным. Это означает что за один раз тип выводится для конкретного выражения. Также стоит помнить что для вывода типов в общем случае требуются ожидаемый тип и выражение в типе которого сожержатся типовые переменные. Тогда мы пытаемся свести тип выражения к ожидаемому типу, в процессе получая ограничения на типовые переменные. Разрешая эти ограничения, получаются значения для типовых переменных. Или ошибка компиляции.

Сначала мы рассмотрим вывод типов описанный в спецификации scala. Пусть $expr$ - это рыражение в типе которого присутствуют типовые переменные. Выделяются три случая:

- $expr.x$ - мы выбираем имя x у $expr$. Тогда вывод типов будет осуществлен для $expr.x$. Это нужно для того чтобы использовать ожидаемое значения или получить информацию от использования x. Например, это позволит написать выражение **Set.empty** $+$ **1** и его тип выведется как **Set[Int]**.

- $expr$ - используется как значение. Тогда нужно найти подстановку типовых параметров которая окажется непротеворечивой. Как именно искать такую подстановку спецификация умалчивает.

- $expr(d_1, ..., d_n)$ - мы применили какие-то подвыражения. Тогда в первую очередь нужно типизировать d_i. В спецификации предлагается два способа: либо заменить использовать типовые переменные как типовые константы, либо, если первый способ не сработал, заменить типовые переменные на $undefined$. Здесь undefined - это специальный для которого правила вывода $\forall T(T <: undefined \wedge undefined <: T)$. Так или иначе, мы типизируем подвыражения. После этого мы можем проверить сводимость их типов к ожидаемым значениям и получить дополнительные ограничения на типовые переменные.

Заметим что в любом случае expr начинается с какого-то метода в который мы не добавили типовые параметры явно.

В Scala Plugin вывод типов устроен иначе. Выше уже говорилость что в процессе сведения мы получаем ограничения при котором сводимость выполняется. Для этого используется тип **UndefinedType**. Он удовлетворяет правилам вывода $undefined$ из спецификации, при этом добавляет добавляет соответствующее ограничение на типовую переменную которую представляет. Однако есть существенное различие. В

спецификации $undefined$ появляется вместо типовых переменных ожидаемого типа, то в нашем случае это типовые переменные выражения.

Важным отличием является отсутсвие отсутсвие стадии замены типовых переменных на типовые константы. В плагине типовые перменные сразу заменяются на **ScAbstractType**. Этот тип похож, в каком-то смысле, на $undefined$. **ScAbstractType** хранит в себе ограничения на соответствующую типовую переменную, доступные из контекста. Например, в коде 10 в процессе вывода типа выражения **magic**, тип **T** будет представлен как **ScAbstractType** и его нижней границей будет **Int**.

Listing 10: Пример ScAbstractType

```
def id[T](t: T): T = t
def magic[U]: U = throw new Exception("no_magic")
val i: Int = id(magic)
```

В процессе сводимости эта дополнительная информация используется для того чтобы прервать заведому бессмысленную проверку. Так, если **ScAbstractType** ограничен сверху классом **Derived**, а мы пытаемся свести к нему **Base**, то это не даст разумного результата. Так же информация о границах вносит существенный вклад для ограничений получаемых **UndefinedType**.

Все информация связанная с типовыми переменными собирается с помощью этой пары: **UndefinedType** и **ScAbstractType**. Для них были добавлены соответсвующие правила вывода.

В случае если проверка сводимости завершилась успешно то **Conformance** возвращает **ScUndefinedSubstitutor** хранящий ограничения для всех типовых переменных встреченных во время проверки. После проверки сводимости всех аргументов Следующий шаг - проверка разрешимости ограничений на типы. Она выполнено довольно просто. Для каждой типовой переменной хранится множества верхних и нижних границ. Сначала находится наименьший тип лежащий выше всех ограничивающих снизу типов. После, аналогично, находится наибольший тип лежащий ниже всех ограничивающих сверху типов. Проверяется их непротеворечивость.

Инструментация сохраняет данные о границах. В зависимости от вариантности добавляет значения для типовых перменных не встреченных в проверках сводимости. А после добавляет эту информации к проверяемой функции.

Во время визуализации информация про выведенные типы попадает в контекст **RelationContext**, который был коде 8 и коде 9. Эти данные необходимы для большей наглядности.

Картинка

3.4. Разрешение перегрузок функций

Последняя часть это выбор правильной перегрузки.

В спецификации описана так.

Реализуется совсем странно.

Использует сводимость.

Заключение

Я бы не советовал в продакшн, много избыточного кода.

Предложена возможность обработки на уровне исходного кода.

Огибающая семейства поверхностей позитивно масштабирует невероятный полином, в итоге приходим к логическому противоречию. Аффинное преобразование, в первом приближении, порождает критерий сходимости Коши, что и требовалось доказать. Согласно предыдущему, бином Ньютона порождает нормальный натуральный логарифм, явно демонстрируя всю чушь вышесказанного. Замкнутое множество позиционирует предел последовательности, что несомненно приведет нас к истине [12]

Список литературы

[1] Author. Что-то про scala native.

[2] Author. Что-то про scalajs.

[3] Author. Что-то про автоматические доказательства.

[4] Author. Что-то про алгебраичные типы данных.

[5] Author. Что то про эта-расширение.

[6] Author. спецификация скала.

[7] Author. что-то про реализацию алгоритма хиндли-милнера.

[8] Oderskiy. Ссылка на dotty, например.

[9] Oderskiy. Статья про DOT систему типов.

[10] Козлов. Дипломная работа.

[11] Пирс. Теория типов.

[12] Стругацкий А.Н., Стругацкий Б.Н. Понедельник начинается в субботу / Под ред. Иванов. — М. : Детская литература, 1965.