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

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

На данный момент scala является достаточно известным языком программирования интерес к которому быстро растет. Одной из важных особенностей scala является его желание элегантно объединить объектно-ориентированное программирование с, набирающей популярность, функциональной парадигмой.

В данной работе нас будет интересовать функциональная направленность scala. К ней можно отнести: функции высших порядков, использование механизма pattern matching или отдание предпочтения неизменяемым данным. А также использование достаточно продвинутой системы типов.

Можно отметить следующие особенности системы типов scala:

- Чтобы найти применение в промышленном программировании scala изначально создавался как язык совместимый с java. Это ведет к возможности использования классов java, а следовательно и типы из java являются допустимыми типами внутри scala.

- Унифицированная система типов. Все типы, начиная с типов функций или классов и заканчивая типами пришедшими из java входят в единую иерархию типов. Это отличается от того же java где ссылочные типы отделены от примитивных. Наибольшим типом в этой иерархии является **Any**, а наименьшим **Nothing**.

- Стандартным методом достижения переиспользования кода в функциональных языках является различного рода полиморфизм. В scala есть возможность как абстрагировать по типу части кода, например полиморфные методы, так и сами типы. К последнему случаю относятся конструкторы типов и параметризованные типы. В случае с конструктором типа есть возможность указать не только ограничения на передаваемые в него типы, но и вариантности. От них зависит как сводимость типовых аргументов скажется на сводимости полных типов. Например, **List[Derived]** сводится к **List[Base]**, если **Derived** сводится к **Base**.

- В scala можно абстрагироваться не только по полноценному типу, но и по конструктору типа. Это дает поддержку так называемых типов высших кайндов.

- Пожалуй, главной особенностью scala является механизм implicit у которого есть множество различных применений.

  Например, для метода часть его параметров можно пометить как неявные. В таком случае можно не передавать соответствующие аргументы явно. Компилятор сам выполнит поиск подходящих по типу неявных переменных, а после использует их в качестве аргументов. В комбинации с типами высших кайндов это позвоялет достичь ad-hoc полиморфизма, когда новая функциональность добавляется в класс без изменения самого класса.

- Также есть другие сущности пришедшие из теории типов [16]: структурные типы, экзестенциальные типы, типы завиящие от пути...

В сложности системы типов есть как плюсы, так и минусы. С одной стороны это повышает надежность кода кода за счет увеличения статичеких проверок, приводя к, так называемой, типобезопасности. Однако вместе с этим усложняется использование языка, а описание типовых конструкций может стать громоздким. Чтобы бороться с последним, компилятор предлагает не только выполнять проверки типов, но выводить большинство типов самомтоятельно.

Часто нет необходимости явно указывать тип переменной, компилятор может сам его вывести используя выражение в правой части. Также может быть необязательно явно передавать типовые аргументы в полиморфную функцию, компилятор может сам сделать неявное преобразование.

Однако все что связано с типобезопасностью компилятор делает неявно для пользователя. А в случае несовпадения типов где-то, он просто выводит сообщение вида: тип ожидался, тип найден. И далеко не всегда просто понять в чем именно заключалась ошибка. Нас будет интересовать возможность как проиллюстрировать процесс выведении типов, так и процесс проверки типов в общем.

Как будет видно в дальнейших, инструменты связанные с отладкой типов часто требуют специальные версии компилторов для своей работы. Заметим что интегрированные среды разработки должны выполнять ту же работу по работе с типами что и компилятор. Действительно, нахождение семантических ошибок требует знания о типах. Поэтому отладчик типов можно основывать не на компиляторе, а на ИСР. А создание подобного инструмента в уже существующей среде, цель которой облегчить написание кода, кажется разумной идеей. Существуют две популярные ИСР для scala: Scala Plugin для intellij idea и плагин scala IDE для eclipse. Далее мы будем говорить про Scala Plugin в intellij idea.

1.1. Отладчик типов для языка программирования OCaml

В качестве примера отладчика типов рассморим проект Type Debugger для языка программирования OCaml.

OCaml, как и большинство функциональных языков, использует мощную статическкую типизацию, и, так же как в scala, компилятор берет на себя работу по выводу типов не указанных явно. Представим ситуацию, во время вывода типов компилятор OCaml находит противоречие: одно подвыражение требует один тип, а другое предоставляет отличный. В таком случае принятие решения о том какой тип на самом деле подразумевался пользователем становится проблематичным. Поэтому компилятор выбирает тип как-то и выдает сообщение об ошибке.

Type Debugger, призван помочь разработчикам на OCaml получать более конструктивную информацию об ошибках во время вывода типов. В ситуациях несовпадения типа он начинает задавать вопросы пользователю, какой тип подразумевался у того или иного выражения. По результатам ответов на эти вопросы Type Debugger понимает источник проблемы и дает пользователю более конкрентые источник ошибки.

Отдельно стоит заметить, что для реализации подобной функциональности Type Debugger для OCaml переиспользует код вывода типов в компиляторе OCaml.

Подобный проект создает прецедент существования отладчиков типов.

1.2. Scala type debugger

Возникает лаконичный вопрос - существует ли подобный инструмент для языка программирования scala. Ответ - да, существует. Проект называется scala type debugger, и для ознакомления с ним можно посмотеть репозиторий на github [1] и несколько статей [2] [3], одна из которых написана в соавторстве с Мартином Одерски, создателем языка scala.

Цель этого инструмента анализировать проблемы связанные с типами. Результатом его работы будет граф, где в вершинах записаны действия выполняемые компилятором при типизации какого-то выражения, а также их результат. По этому графу можно понять что делал компилятор для проверки типов и в какой именно момент возникла ошибка.

Так как это прямой конкурент, то разберем что именно нас не устраивает в данном проекте:

- Так же как и Type Debugger для языка OCaml, scala type debugger переиспользует код компилятора scala. Для своей работы он требует специально инструментированную версию scalac. Это не очень удобно, нужно перенастраивать окружение под соответсвующую версию компилятора. При этом нашей целью является создать отладчик типов внутри Scala Plugin. Scala Plugin же занимается анализом кода самостоятельно и использует свое собственно внутреннее представление типов. Добавление некоего инструментированного компилятора создаст лишнюю зависимость.

- На момент написания работы репозиторий со scala type debugger не обновлялся в течение 5 лет. В статье [2] написано о планах интегрировать его в eclipse. Но до сих пор этого не было сделано.

1.3. Show implicit parameters action

В заключение рассмотрим уже существующий в Scala Plugin инструмент, который помогает пользователю разобраться в дейтсвиях компилятора.

В начале раздела уже описывалась работа механизм implicit параметров. Существуют определенные правила поиска соответвствующих параметров, но так как в коде это явно не указываются, часто этот механизм может работать не вполне предсказуемо. Ситуация осложняется существованием функций производящих неявные значения используя другие неявные параметры и так далее.

Чтобы помочь облегчить работу с implicit парметрами в Scala Plugin существует show implicit parameters action [14]. Он в виде вложенных вкладок отрисовывает дерево, где узел - это функция требующая неявные параметры, а лист - это неявная переменая. Это дерео позволяет быстро навигироваться по коду.

Вообще говоря, многие процессы удобно визуализировать как деревья. Забегая вперед, в данной работе будет использоваться такой-же стиль представления информации как и у show implicit parameters action.

1.4. Постановка цели

На данный момент мы узнали достаточно чтобы сформулировать цель и задачи.

Цель: реализовать отладчик процесса проверки типов в Scala Plugin.

Для этого можно выделить следующие задачи:

- Инструментировать Scala Plugin для сбора данных о процессе работы с типами.
- Дать интерпретиацию приближенную к спецификации.
- Минимизировать влияние инструментации во время выполнения на оставшуюся часть плагина.

Теперь несколько замечаний про цели и задачи.

Несмотря на то что цель звучит достаточно общно, в разделе 3 будут конкретизированы процессы работы с типами вызывающие интерес.

В разделе 2 будет более подробно рассказано почему в качестве способа сбора данных было выбрано именно инструментирование исходного кода плагина. Там же описана важность уменьшения влияния инструментации во время исполнения.

Также не получится целиком давать интерпретацию данных основываясь на спецификации потому что плагин порою ей не следует.

Можно привести три случая непостредственной пользы подобного инструмента:

- Понизить сложность вхождения. Как говоилось выше, у scala не самая простая система типов и разобраться в ней новичку может быть не просто. Данный ин-

струмент позволит явно визуализировать работу с типами, которую производит Scala Plugin.

- Нетривиальные случаи. Хотя принцип работы будет основан на коде Scala Plugin а не scalac, существуют неочевидные ошибки не завязанные на специфике компилятора с которыми инструмент поможет спраиться. В разделе 3.4 будет дан пример такой ситуации.

- Внутреннее использование. Для людей которые должны работать с внутренним устройством плагина это может дать представление о происходящем внутри без необходимости непосредственной отладки.

2. Особенности реализации

В этом разделе мы будем говорить о получении данных из 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, скорее всего, связанной с типами. 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. 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 производительность 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 функций будет расммотрено в разделе 2.3, а распространение на классы в разделе 2.4.

2.1. Инструментирование

Здесь мы составим набор правил для использования инструментации так, будто макросов нет. Это позволит нам запускать Scala Plugin 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, например, 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**, как в примере кода 1.

Пример кода 1: Явная передача объекта инструментирования внутрь функции

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

Монаду **Option** с объектом через который будет производиться инструментирование мы будем называть объектом инструментирования. Когда нужно будет солаться на сам объект, будет употребляться слово непостредственный объект инструментирования.

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 непосредственный объект инструментирования, и которые игнорируются если **Option** оказался пуст. Это показано в примере кода 2.

Пример кода 2: Действия внутри контекста инструментирования

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

Далее мы будем говорить про **Option** как о контексте инструментирования. Соответственно, объект инструментирования создает контекст инструментирования, в кото-

ром с помощью функций высшего порядка мы можем выполнять какие-то действия, и который может оказаться пуст.

Теперь поймем как хранить данные которые изменяются со временем. Понятно что мы можем их хранить внутри объекта инструментирования, 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 контекстов 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. Для этого можно явно связать переменную с объектом инструментации, в примере кода 3 показано как это сделать используя функцию **map**.

Пример кода 3: Привязка к контексту инструментирования

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

В нем мы явно связали переменную **intermediate** с контекстом инструментирования объекта **instrumentation**. Тем самым мы породили дополнительный контекст инструментации. Аналогичным приемом можно воспользоваться если мы захотим создать дополнительный объект инструментирования. Это может понадобиться, например, для вложенных вызовов, как в примере кода 4.

Пример кода 4: Создание нового объекта инструментирования

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

Таким образом новые контексты инструментирования можно создавать явно, используя уже существующие через функцию **map**.

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 аргументов совпадали и для каждой пары соответствущих типовых аргументов выполнялось определенное соотношение. При проверке этих соотношений достаточно дойти до первого ложного чтобы понять что сводимость нет,

что Scala Plugin 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 включенном инструментировании, 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 контекста инструментирования в связке с логическим или, а также условия 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 кода 5 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.

Пример кода 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 послужить **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 макрос.

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-либо проверок. 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 код.

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 инструментирования 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.

2.3. Макросы для функций

Сформулируем правила для инструментирования которое обсуждалось 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 инструментирования происходит явно от другого, уже существующего, контекста инструментирования. Для этого подходит функция функция **map**.

4. Для того чтобы влиять на первоначальную логику добавляются инвариантные для пустого контекста инструментирования 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.

2.3.1. Поиск инструментирования

Алгоритм 1 Обновление информации о контекстах инструментирования

```
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:     return names
```

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

Эта функция нужна для обновления информации об именах связанных с инструментированием. Входными данными является: блок кода образующий новую область видимости и имена из вышележащей области видимости. Требуется найти в блоке кода имена с новыми контекстами инструментирования, а также убрать имена затененные. Это не сложно сделать, так как по пункту 3 связь между новыми и старыми контекстами отслеживается в коде явно. Новые имена получаются из старых при помощи функции **map**, остальное затенение.

2.3.2. Генерация неинструментированной функции

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

Теперь перейдем к рассмотрению генерации неинструментированной версии функции.

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 в пункте 1. 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 инструментиорвания в условиях. Чтобы вернуть все как было достаточно упростить все логические выражения, подразумевая что используемые в них контексты инструментирования пусты. А также удалить условные конструкции, если они зависят только от объектов инструментирования.

2.3.3. Генерация инструментированной функции

Алгоритм 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: function HandleInstrumentedBlock(codeBlock, previousNames)
6:     names ← UpdateInstrumentationNames(codeBlock, previousNames)
7:     for expr ← выражения внутри codeBlock do
8:         if expr это вызов с использованием элемента из names then
9:             добавить к имени символа суффикс $I
10:        else if expr это новый блок кода then
11:            HandleNonInstrumentedBlock(expr, names)
```

Логика генерации инструментированной функции намного проще. Здесь нам достаточно изменить название самой функции и проследить чтобы все вызовы, в которые были переданы контексты инструментирования, были перенаправлены в инструментированный код.

2.3.4. Анализ предложенного решения

Предложенное решение вернет функцию **f** к изначальному виду, тем самым решая поставленную задачу. Избыточность создается только из-за увеличения количества функций доступных для виртуальной машины java.

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 об их наличии.

2.4. Макросы для классов

Теперь перейдем к рассмотрению работы макроса на классе. Нашей задча остается прежней - по классу **C** сгенерировать пару: инструментированный класс и неинструментированный.

Прежде всего заметим, что все относящееся к обработке инструментирования внутри функции также верно и для класса. Здесь нас будут интересовать проблемы при создании классов, а также возможности 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 по ссылке **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 перестанет 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 должно было быть инструментирование, то нам нужно перезапустить конструктор в классе **$I**, но уже с ним. Это может привести либо к несогласованности внутри класса, либо, если есть побочные эффекты, к неожиданным внешним

проявлениям. 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 2, нас интересуют классы являющиеся упаковками функций. В них отсутсвует секция инициализации.

Теперь нужно разобраться с переменными класса. Если нет инициализации, то публичные переменные класса **C** мы можем просто переиспользовать в методах класса **$I**. Приватные же переменные класса **C** не покидают его область видимости, поэтому их возможно пересоздать. 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 другими именами, а потом произвести соответсвующие присвоения.

2.4.4. Изменение методов родителя

Последняя рассмотренная здесь сложность будет вызвана инкапсуляцией. 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 у 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**.

2.4.5. Анализ предложенного решения

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 java и может делать невозможными некоторые отимизации.

Пример кода 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
  }
}
```

Также для наглядности приведен примеры кода до обработки макросом 6 и после 7.

Пример кода 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
    }
  }
}
```

На этом часть связанная с инструментированием кода и макросами заканчивается. Все вышеописанное можно посмотреть в коде. 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.

3. Реализация

В этом разделе наше внимание сместится на особенности реализации процессов связанных с проверкой типов 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 [10]. Мы проследим как 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 для каждого процесса будет приведен пример его визуализации.

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.

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 рассмотренный в разделе 1.3, и **GoToImplicitConversationAction**, помогающий в работе с неявными преобразованиями. Так же в данной работе не освящена работа с динамическими типами. В будущем эти пробелы можно восполнить.

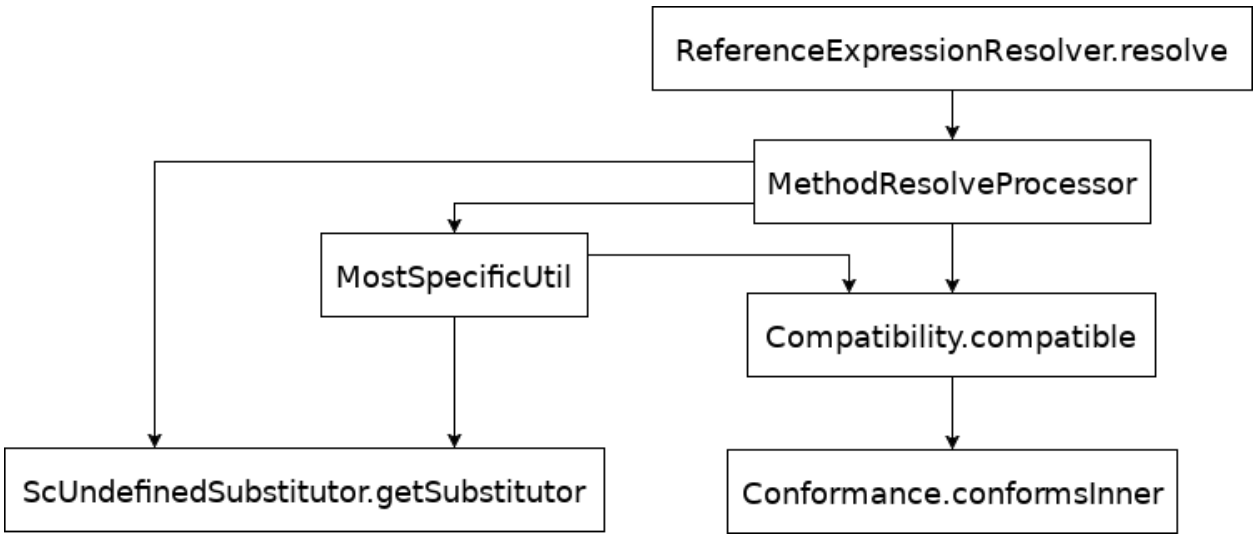


Рис. 1: Граф вызовов

3.1. Архитектура Scala Plugin

Как говорилось в начале раздела 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. На рисунке 1, несколько упрощенно, показан граф вызовов такой обработки. Мы явно будем указывать классы, основные и вспомогательные функции в которых использовалась аннотация **uninstrumented** из раздела 2. Это поможет дучше понять куда именно добавлялась инструментация.

Представим что у нас есть применение символа к набору выражений-аргументов. Для обработки этого применения у объекта **ReferenceExpressionResolver** будет вызван метод **resolve**. 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**:

- **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 сводимости двух типов используется интерфейс **Conformance**. У этого интерфейса две реализации: одна для системы типов scala, другая для системы типов dotty [12], 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 проверки сводимости написано в разделе 3.2. Аннотировано **uninstrumented**:

- **Conformance.conformsInner**
- **Conformance.computable**
- **Conformance.checkParameterizedTypes**
- **Conformance.addParam**
- **Conformance.addArgedBound**
- **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 *compound type*. 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 [11].

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**:

- **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 сравнение типов используемых 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 [6], 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. Для этого у 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 повторения 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 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**, который должен быть параметризованным **scala.Array**. Отдельная сущность для массива влечет дублирование кода для параметризованных типов, в котором и так очень много повторений. Также другия несоответсвия в типах. Все это сильно повышает неоднородность кода.

3.2.1. Визуализация

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 способа представления мы будем использовать вложенные вкладки, которые имеют древовоидную структуру. Это достаточно естественное решение, учитывая что само сведение является представляется 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.

Пример кода 8: Узел условия

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

Пример кода 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))
}
```

В обоих случаях во время проверки используется класс **RelationContext**. Это нужно чтобы перадть данные полученные во время других процессов.

В ходе работы собираются условия. После рекурсивных вызовов полученные условия объединяются в отношения. Мы получаем уже, по сути, готовую для визуализации структуру. Посмотреть пример результата можно на рисунке 2. Далее мы не будем подробно останавливаться на визуализации. Достаточно знать что она всегда строится примерно одинаково и представляется в виде дерева.

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 Plugin и спецификацией scala. Также будет написано как правила вывода из спецификации переносятся в плагин. Здесь будет дана очень общая информация о типах, чтобы получить более подробную информацию, например про синтаксис, рекомендуется посмотреть в спецификацию.

3.2.2. 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 [8], а апогеем этой идеи является DOT calculus [13].

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. Проверка сводимости для *singleton type* заменяется на проверку эквивалентности. Аксиома о сводимости типа **Singleton** в плагине отсутвует.

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.

3.2.3. 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, как правило являются *type*

projection. 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. Для него правила вывода *type projection* добавляются без особых сложностей.

3.2.4. Type Designator

Для того чтобы не писать все время **scala.type#Int**, как в предыдущем пукте, существует *type designator* являющийся коротким синонимом 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 абстрактных типов, но про это будет написано в разделе Abstract Types.

3.2.5. Parameterized Type

Parameterized type появляется когда у нас есть конструктор типа и мы хотим применить его к типовым аргументам. Например, **List[Int]** - это *parameterized type*, где **List** - это конструктор типа, принимающий один типовой аргумент. Для типовых параметров возможны три вида вариантности: ковариантность, контрвариантность и инвариантность. Варинатность влияет на сводимость *parameterized type*. Ковариантность говорит что параметры должны сводится так же как *parameterized type*, контрвариантность что наоборот, а инвариантность что должны совпадать.

В плагине есть соответствующий класс **ParameterizedType**. Понятия и правило вывода из спецификации переносятся на него, хоть логика работы с ним и тяжеловесна.

3.2.6. Compound Type

Compound type объединяет в себе понятия конъюнктного и структурного типов. Идейно, можно представлять что конъюнктный 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. Так **AnyRef { def f(): Int }** - это просто тип у которого есть метод **f**, возвращающий **Int**. Если какой-то тип доходит под это определение, то он сводится **AnyRef { def f(): Int }**. Если объединить эти две концепции, то получится *compound type*.

В Scala Plugin *compound type* представлен классом **ScCompoundType**. Логика работы с ним соответсвуют описанным в спецификации правилам введения и выведения, которые и требуется добавить. 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.

3.2.7. Existential Type

Existential type - это просто экзестенциальный тип, подробнее о котором можно почитать в [16]. 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 содержится информация об имени и границах типа, а также типовых агрументах. Последнее нужно для поддержки типов высших кайндов. Правила ввода и вывода можно брать из спецификации. 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.

3.2.8. Method Type

Как известно, в scala функции являются объектами первого порядка. Иначе говоря, мы можем передавать функции как аргументы, сохранять в 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 [9].

В плагине используется есть соответсвующий класс **ScMethodType**. Так как *method type* можно свести только к *method type*, его обработка локализована и в точности повторяет то что написано в спецификации. Поэтому правило сведения можно переносить напрямую.

3.2.9. 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 правило вывода из спецификации.

3.2.10. 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**. Поэтому это не добавляет нового правила вывода.

3.2.11. 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. Для работы со всеми ними можно использовать аксиомы для работы с *abstract type*.

3.2.12. 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. Но это не обязательно правда для dotty. В плагине нет типов соответсвующих им, вместо этого он сразу производит удаление синтаксического сахара. Подробнее про это можно прочитать в работе [15].

3.2.13. 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. Правило вывода такое же как у *parameterized type*. Вместо массивов в scala используется конструктор типа **scala.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 [5] или scala native [4].

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, которые и были перенесны.

Так же в плагине появляются классы **ScAbstractType** и **UndefinedType**. Эти типы 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 содержатся 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 выражение 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 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.

В любом случае вывод типов начинается с какого-то полиморфного 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 необходимые чтобы сводимость выполняется. Для этого используется тип **UndefinedType**. Он удовлетворяет правилам вывода *undefined* из спецификации, при этом 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**.

Пример кода 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. 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.

Пример можно посмотреть на рисунке 3.

3.4. Разрешение перегрузок функций

Последним рассматриваемым нами процессом будет выбор перегрузки функции. Как и впрошлый раз, посмотрим как процесс описывается в спецификации scala, а уже потом перейдем к реализации в Scala Plugin.

Допустим что существует несколько объявлений имени **f**. Далее будут описаны несколько стадий, каждая из которых должна отсеять потенциальных кандидатов по какому-либо признаку. Если после какой-то стадии остался один кандидат, то он считается правильной перегрузкой, если ни одного, то это ошибка компиляции. Если в конце кандидат не один, то это тоже ошибка компиляции.

1. В первую очередь вводится понятие *shape* для выражения e по следующим правилам

   - если e - функция $T \Rightarrow b$, то $shape(e)$ это $Any \Rightarrow shape(b)$
   - для именнованного параметра имя сохраняется
   - во всех остальных случаях $Nothing$

   Заметим, что тип выражения e всегда сводится к $shape(e)$. Тогда оставим только тех кандидатов, которые применимы к аргументам, тип которых мы упростили до *shape*.

   Это важная стадия, она позволяет выбрать кандидата не опираясь на конкретные типы аргументов. Это важно, потому что часто для вывода типа аргумента требуется ожидаемое значение, а его не узнать пока мы не выберем кандидата.

2. Типизируем выражения переданные в качестве аргументов без ожидаемого типа. Оставим только тех кандидатов, которые можно применить к этим типам.

3. Оставим только тех кандидатов, которые не используют параметров по умолчанию.

4. Иначе нужно выбрать наиболее специфичного кандидата. Для определения наиболее специфичного кандадта, нужно найти того, кто более специфичен чем все остальные.

   Существет два критерия чтобы понять что один кандидат более специфичен чем другой.

   - Один кандидат объявлен в пространстве имен унаследованном от другого кандидата.
   - Один кандидат сводится к другому кандидату. Для методов это означает что аргументы одного метода всегда применимы к другому.

В Scala Plugin отбор кандидатов происходит в классе **MethodResolveProcessor**. Первые кандидаты уже будут отфильтрованы по *shape*, что хорошо. Иначе бы показывались все варинаты, даже не подходящие по количеству аргументов. Однако, при этом могло не учитываться наличие некорректных именованных аргументов. Реализуется совсем странно.

В первую очередь плагин проверяет применимость кандидатов к переданным в каестве аргументов выражениям. Все это происходит с помощью **Compatibility** и **ScUndefinedSubstitutor**, и описывалось ранее. Инструментация сохраняет данные полученные в результате этих проверок.

Далее идут проверки на неправильные именованные аргументы, а также аргументы со значениями по умолчанию. Их результаты тоже сохраняются.

По спецификации scala остается найти наиболее специфичного кандидата. Это происходит внутри класса **MostSpecificUtil**. Происходящее там по большей части повторяет написанное в спецификации.

Результат работы для перегруженных функций продемонстрирован на рисунке 4.

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 [17]

Список литературы

[1] Author. Статья про scala type debugger1.

[2] Author. Статья про scala type debugger1.

[3] Author. Статья про scala type debugger2.

[4] Author. Что-то про scala native.

[5] Author. Что-то про scalajs.

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

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

[8] Author. Что-то про зависимые типы в scala.

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

[10] Author. спецификация скала.

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

[12] Oderskiy. Ссылка на dotty, например.

[13] Oderskiy. Статья про DOT систему типов.

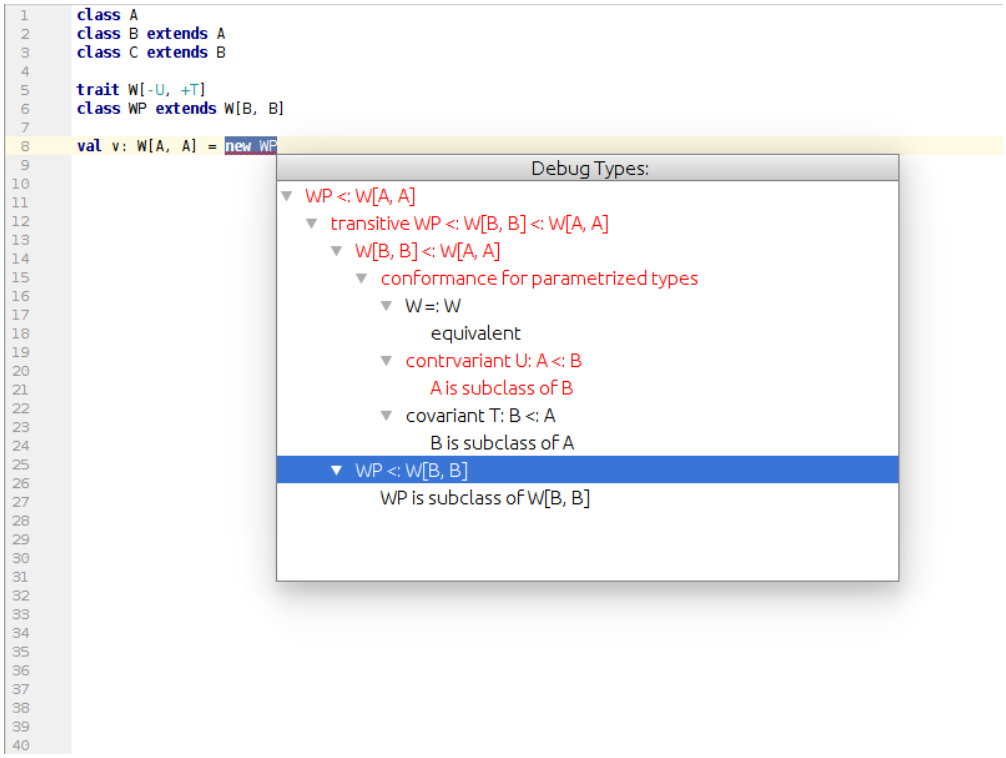
[14] Podkhalyuzin Alexander. Show implicit parameters action.

[15] Козлов. Дипломная работа.

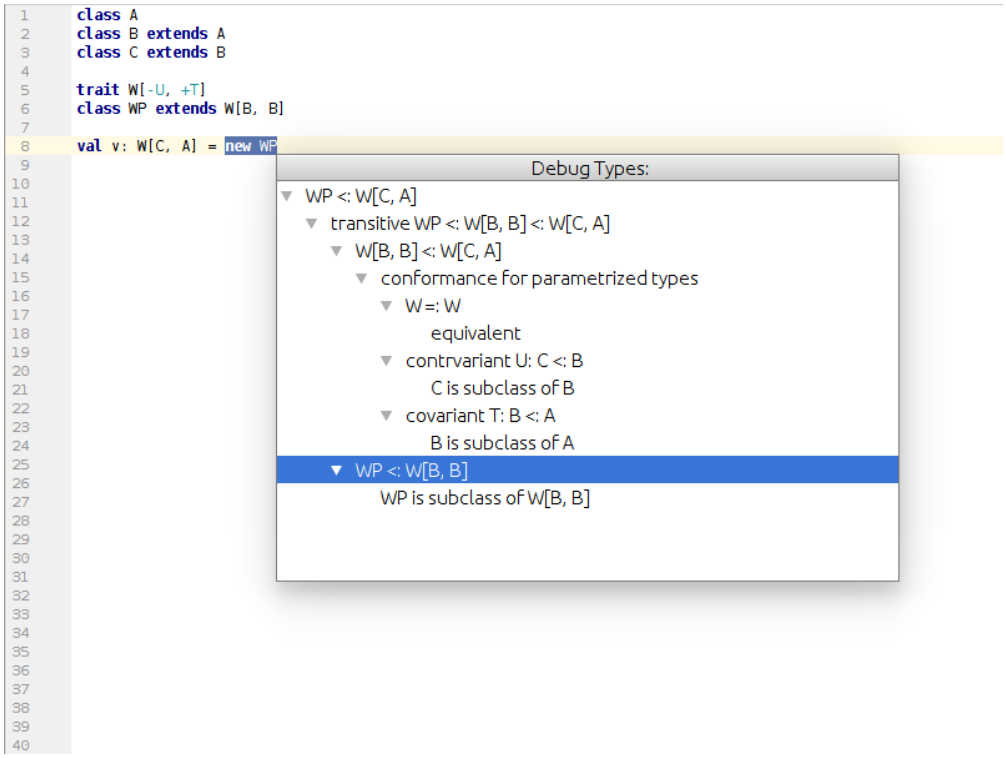
[16] Пирс. Теория типов.

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

Пример проверки сводимости



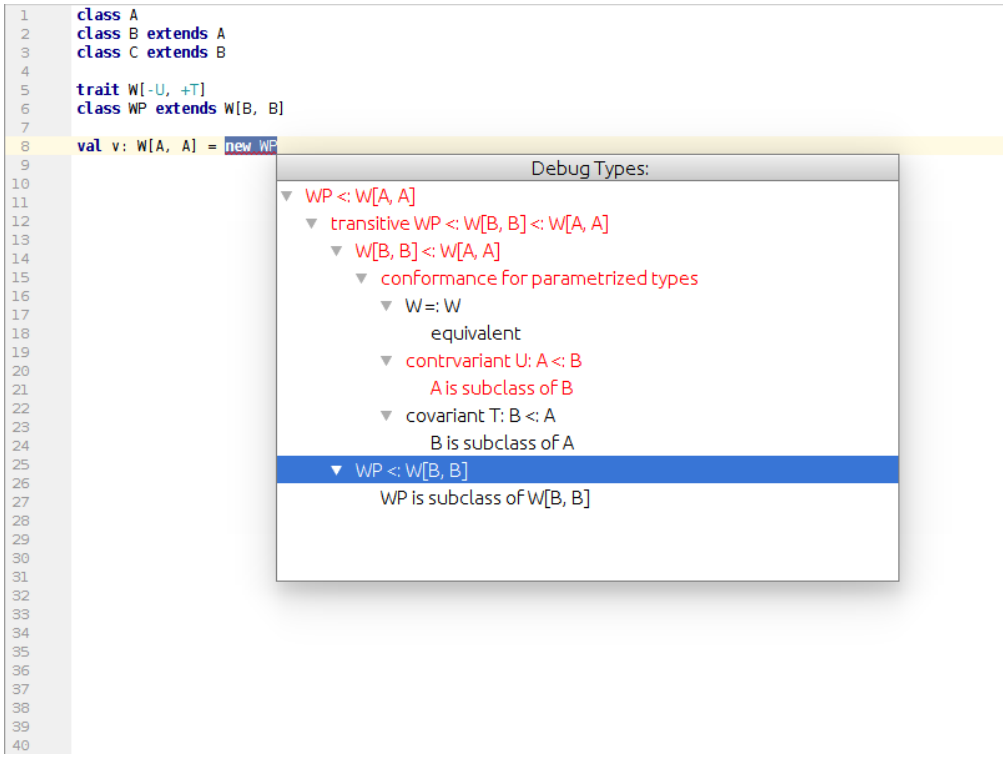
(a) Неправильное сведение



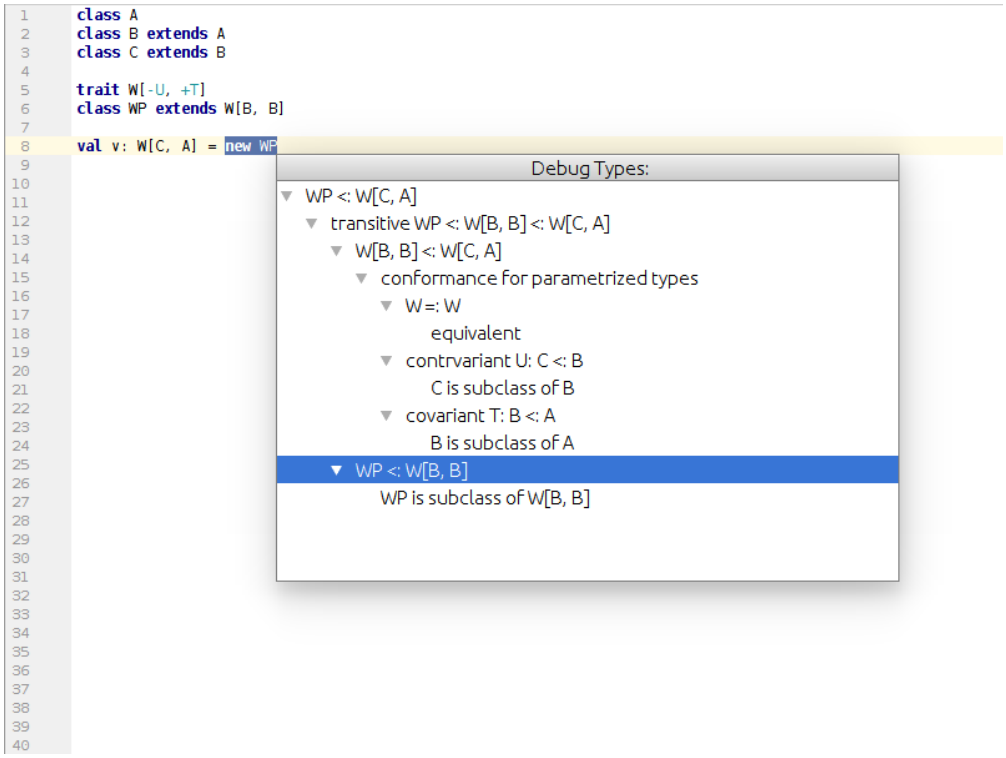
(b) Правильное сведение

Рис. 2: Проверка сводимости

Пример вывода типов



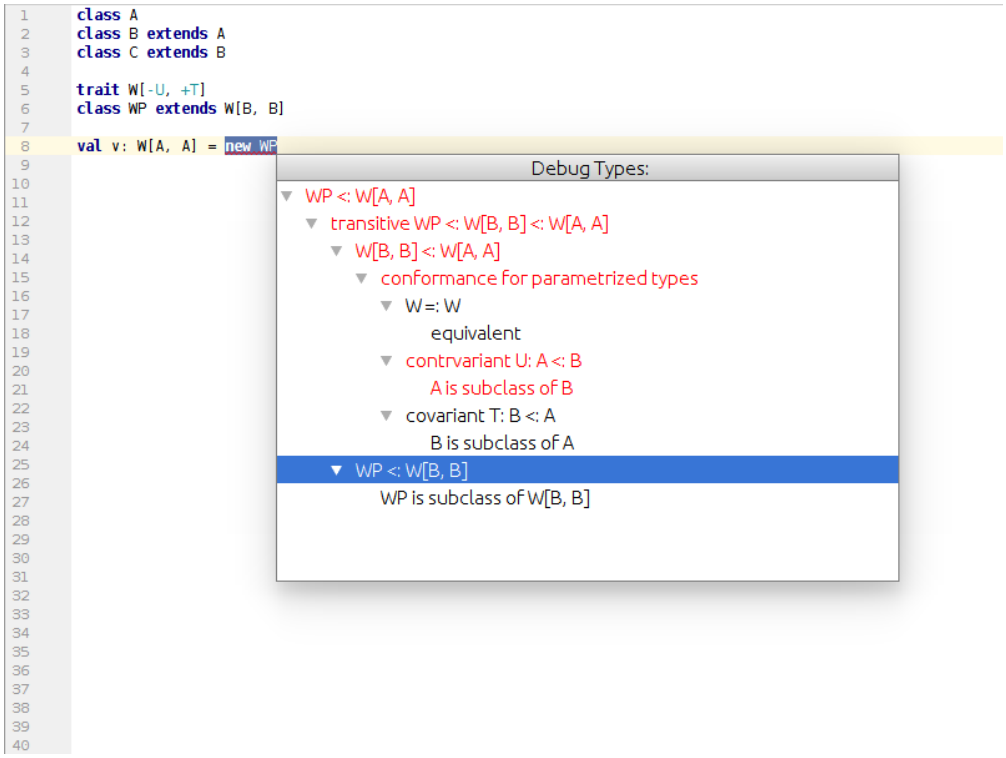
(a) Неправильное сведение



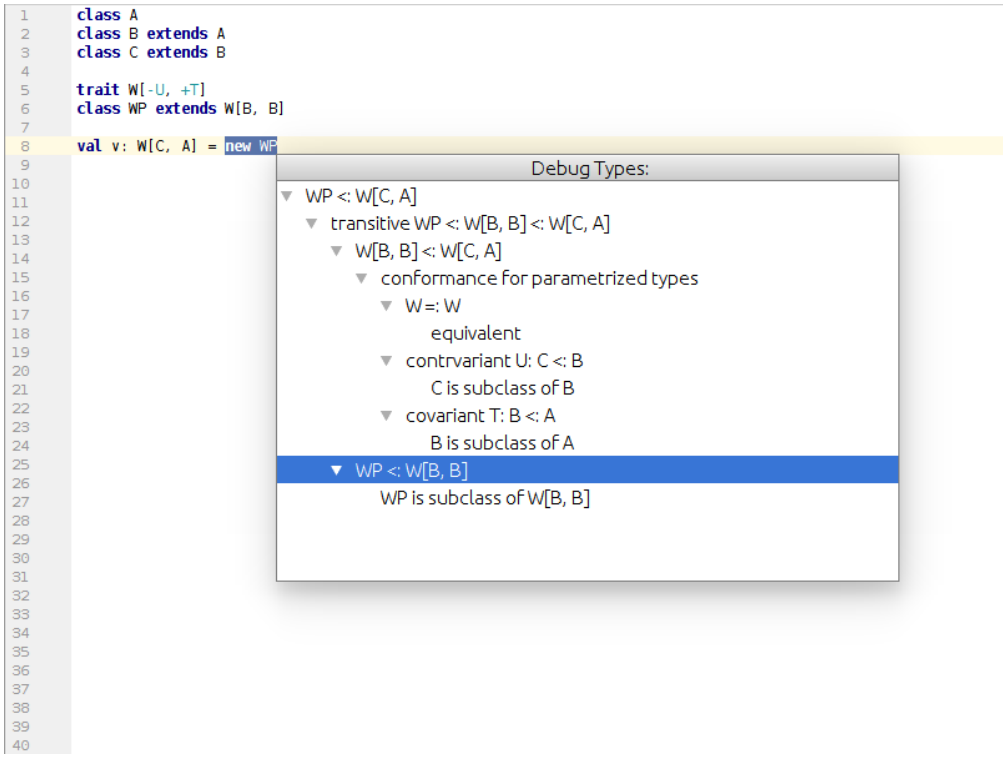
(b) Правильное сведение

Рис. 3: Проверка сводимости

Пример выбора перегрузки функции



(a) Неправильное сведение



(b) Правильное сведение

Рис. 4: Проверка сводимости