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 **4**

1. Обзор **5**

1.1. Отладчик типов для языка программирования OCaml 5

1.2. Scala type debugger 5

1.3. Show implicit parameters action 5

2. Особенности реализации **6**

2.1. Инструментирование 7

2.2. Макросы в scala 9

2.3. Макросы для функций 10

2.4. Макросы для классов 12

2.5. Сохраненные данные и визуализация 14

3. Реализация **15**

3.1. Архитектура Scala Plugin 15

3.2. Проверка сводимости типов 15

3.2.1. Singleton Types 16

3.2.2. Type Projection 16

3.2.3. Type Designators 16

3.2.4. Parameterized Types 16

3.2.5. Compound Types 16

3.2.6. Existential Types 16

3.2.7. Method Types 16

3.2.8. Polymorphic Method Types 16

3.2.9. Type Constructors 16

3.3. Вывод типов 16

3.4. Разрешение перегрузок функций 16

Заключение **17**

Список литературы **18**

Приложение A. Алгоритмы обработки инструментрованных функций **19**

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

1.1. Отладчик типов для языка программирования OCaml

1.2. Scala type debugger

1.3. Show implicit parameters action

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.

```
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.

```
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.

```
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.

```
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 пре-

рываться 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.

```
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.

Обычно эти 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.

Теперь 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.

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.

Теперь перейдем к рассмотрению генерации не инструментированной 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. 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.

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.

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**.

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**.

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.

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.

В случае с классом мы получаем избыточность сразу во многих местах: дополнительный класс, дополнительные методы-заместители, возможно изменение модификатора приватный на защищенный 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. Всего в проекте потребовалось использовать 28 таких аннотаций.

2.5. Сохраненные данные и визуализация

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 [1]. 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 pattern matching.

Если нужно, добавить пример каких-то классов.

3. Реализация

В этом разделе наше внимание сместится на осбенности реализации процессов связанных с типами в Scala Plugin. Особенное внимание будет уделено спецификации scala [2]. Мы проследим как соответсвующие понятия от туда переносятся в Scala Plugin.

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. Перечислим интересующие нас процессы.

В первую очередь нужно научится сравнивать два типа. 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.

Вывод типов - что-то было

Разрешение перегрузок - ...

implicits - уже существует инструмент... Динамические типы...

При этом не стоит забывать что Scala Plugin ровно как и компилятор scala должны реализовывать спецификацию scala [2]. Смысл в последующей визуализации Хочется сделать С уважением к спецификации Однако не везде это возможно сделать Где можно сделаем Везде будет указано расхождение

Достаточно абстрактоно и упрощенно архитектуре плагина будет рассмотрена в разделе 3.1. Там же дается общая ифнормация о сводимости типов, выводе типов и поиске наиболее специфичной перегрузки.

Более подробно о типах scala и их представлении в плагине. В разделе 3.2 будут рассмотрены типы описанные спецификацией scala на примере проверки их сводимости. Также внимание будет уделено отличию от типов в scala plugin.

В разделе 3.3 основной темой будет отличия вывода в плагине от спецификации.

Завершит все разрешение перегрузок 3.4.

3.1. Архитектура Scala Plugin

Постоянные проверки типа Any, Nothing... JavaArray Object

Хиндли-Милнер [3]

3.2. Проверка сводимости типов

Value Types

Non-Value Types

3.2.1. Singleton Types

3.2.2. Type Projection

3.2.3. Type Designators

3.2.4. Parameterized Types

3.2.5. Compound Types

3.2.6. Existential Types

3.2.7. Method Types

3.2.8. Polymorphic Method Types

Типы высших кайндов - куда-то

3.2.9. Type Constructors

Запустить, посмотреть во что выльется.

3.3. Вывод типов

Вывод типов локальный. В спекцификации вывод типов так-то. В плагине абсолютно иначе.

3.4. Разрешение перегрузок функций

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 [4]

Список литературы

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

[2] Author. спецификация скала.

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

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

Алгоритмы обработки инструментированных функций

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
```

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
```