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

Кафедра математических и информационных технологий

Саввинов Д.К.

Разработка системы условных эффектов с реализацией на языке программирования Kotlin

Магистерская диссертация

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

Научный руководитель:
руководитель команды компилятора Kotlin Ерохин С.Е.

Рецензент:
магистр техники и технологии Ахин М.Х.

Санкт-Петербург
2017

Оглавление

Аннотация

На сегодняшний день, статический анализ кода выполняет ряд важнейших функций: автоматизация рутинных действий (таких как приведение типов), поиск простых ошибок, оптимизации кода, рефакторинг.

Статический анализ можно условно разделить на внутрипроцедурный (работающий только над отдельно взятой процедурой) и на межпроцедурный (рассматривающий при анализе всю программу в целом). Вместе с тем, большинство существующих систем не выполняют интенсивного межпроцедурного анализа, что серьезно ограничивает полноту результирующей диагностики.

Данная работа посвящена изучению одного из возможных подходов к межпроцедурному анализу, основанного на учете т.н. «побочных эффектов» (англ. *side-effects*) функций. Значительное внимание было уделено «условным эффектам» – т.е. эффектам, которые вызываются только при исполнении некоторых условий. Было введено понятие «схемы эффектов» – краткого перечисления всех возможных условных эффектов функции. На основе этих понятий была разработана система условных эффектов.

В качестве демонстрации возможностей предложенной системы, ее прототип был реализован на практике и успешно использован для улучшения статического анализа в языке `Kotlin`.

Введение

Актуальность задачи статического анализа кода, на сегодняшний день, уже не вызывает ни у кого сомнения. Статический анализ обнаруживает ошибки в коде (как связанные с логикой, так и связанные с быстродействием, стилем), поставляет информацию, необходимую для проведения оптимизаций в коде, его рефакторинге. Статические анализаторы используются как в узкоспециализированных инструментах, основной задачей которых является непосредственный анализ кода, так и в качестве составной части других, более крупных систем – таких как компиляторы, IDE, оптимизаторы.

Одним из возможных методов статического анализа являются **системы эффектов**. Эффект – это некоторое воздействие вычисления (функции, метода, подпрограммы) на состояние вычислителя. Простым примером эффекта является запись или чтение в разделяемую подпрограммами переменную, или же тот факт, что функция может бросить некоторое исключение (последний тип эффектов был реализован в языке `Java` в виде механизма проверяемых исключений). Система эффектов отвечает за корректный учёт всех эффектов, их взаимодействие друг с другом, а также другие действия с ними (вывод, проверка, и т.д.)

Важным свойством, которому будет уделено значительно внимание, является то, что при анализе кода эффекты часто *наблюдаются*. Так, в блоке `catch` в `Java` мы наблюдаем тот эффект, что некоторая функция сгенерировала исключение. Однако в классических системах эффектов, это наблюдение не дает анализатору никакой информации. Вместе с тем, если бы было известно, какие условия вызывают такой эффект, то из факта наблюдения можно было бы сделать вывод, что окружение удовлетворяет поставленным условиям, тем самым извлекая больше информации из кода.

В связи с этом, мы предлагаем систему эффектов, расширенную добавлением условных эффектов, про которые известны вызывающие их условия. Это позволяет описать в рамках системы более широкий набор функций и соответствующих им эффектов, и, что наиболее важно, извлечь из использования системы значительно больше пользы при тесной интеграции с компиляторами и/или IDE.

В качестве демонстрации применения предложенной системы, будет изучена польза от рассмотрения условных эффектов на примере языка `Kotlin`. Данный язык интересен из-за механизма «умных приведений типов» (англ. *smartcasts*) – если в некотором участке кода статически гарантированно, что переменная имеет более частный тип, то компилятор сделает приведение к этому типу автоматически. Этот механизм естественным образом дополняет систему условных эффектов: пронаблюдав эффект и узнав некоторую типовую информацию о переменных в окружении, можно сделать умное приведение типов там, где раньше оно не делалось или же было вовсе невозможно.

Таким образом, в работе будет рассмотрен ряд эффектов, естественным образом возникающих из предметной области – в частности, эффекты, сообщающие о типе или значении переменных окружения. Кроме того, в работе будет показано, как ввести эффекты, сообщающие о том, что некоторая процедура была вызвана детерминированное количество раз. Данные эффекты могут быть очень полезны в комбинации с другими видами анализа, нуждающимися в подобных гарантиях для сохранения консервативности (например, анализ инициализации переменных).

Также будут рассмотрены вопросы комбинации условных эффектов при вложенных вызовах. Отдельное внимание будет уделено проблеме работы с эффектами в присутствии частичных вычислений.

В заключение, будет произведен анализ применимости систем условных эффектов, будут намечены направления развития данного исследования.

1. Анализ предметной области и существующих решений

1.1. Обзор существующих решений

1.1.1. Системы эффектов

Системы эффектов появились в конце 80-ых годов как логичное развитие систем типов. Из-за этого, между системами типов и системами эффектов существует естественная связь, служащая источником для различных интуиций и полезных параллелей. Поскольку осознание этой связи значительно облегчает понимание любых фактов и свойств систем эффектов, ниже будет вкратце описано, откуда эта связь появляется.

В компьютерных программах существует два основополагающих понятия: это объекты, которые хранят информацию, и функции, которые что-то делают с объектами

Системы типов отвечают за информацию об объектах. Мы можем написать нечто вроде `a: Int`, и это будет означать, что в этой переменной хранится целое число.

Но мощь систем типов ограничена, когда речь заходит о функциях – что и неудивительно, т.к. тип говорит о свойствах объекта, а не процедур. Отчасти проблема решается сигнатурами, но по сути, это всего лишь информация об аргументах и возвращаемом объекте, но никак не о том, что с ними делает функция. Так, сигнатура `void doSomething()` в `Java` не дает никакой информации, и программисту остается лишь надеяться на адекватное название и комментарии в коде.

Отсюда и рождается идея эффектов и систем эффектов. Эффект для процедуры – это тоже самое, что тип для объекта. Так, в одной из первых работ по системам эффектов, дается следующее определение:

Определение 1.1. Эффектом выражения называется краткое описание всех наблюдаемых побочных эффектов, которые это выражение может вызвать при вычислении [15].

Традиционно для подобных исследований, авторы немного лукавят и не определяют «побочный эффект» строго, надеясь на интуицию читателя (в их оправдание скажем, что интуиция, скорее всего, не подведет). В главе 2 мы аккуратно разберемся со всеми этими понятиями и дадим точные определения. Пока же можно считать, что «побочный эффект» соответствует некоторому изменению в памяти программы.

Тесная связь между системами типов и системами эффектов сходу дает значительное количество полезных наблюдений. Чтобы подчеркнуть двойственность между этими двумя системами, эти наблюдения оформлены в таблицу 1

Системы типов	Системы эффектов
Помогают предотвращать ошибки – например, от сохранения строкового значения в тип `Int`.	Помогают предотвращать ошибки – например, от использования функции, которая бросает исключение, вне блока `try-catch`.
Помогают компилятору – например, знание о типе переменной может позволить хранить ее в памяти более оптимально.	Помогают компилятору – например, знание о том, в какие переменные пишет функция и какие читает, может позволить переупорядочивание кода, ранее невозможное.
Типы/эффекты могут быть либо явно написаны пользователем, либо выведены автоматически	
Бывает полезной возможность введения пользовательских типов/эффектов	

Таблица 1: Сравнение типов и эффектов

Основополагающей работой является уже упомянутая несколько раз статья Lucassen, Gifford «Polymorphic effect system» (1988) [15]. В ней авторы описывают основы систем эффектов, предлагают формализм, схожий с формализмом систем типов, позволяющий описывать эффекты. Данное исследование является в большей степени математической формализацией концепта эффектов, нежели руководством к тому, как использовать этот концепт на практике.

С точки зрения практики, очень интересной является статья Greenhouse, Boyland «An Object-Oriented Effect System» (1999) [9]. В ней авторы задаются уже вполне жизненными и насущными вопросами о применении систем эффектов в условиях объекто-ориентированного `Java`-подобного языка. Так, рассмотрены проблемы наследования эффектов, инкапсуляции в присутствии эффектов.

Следует отметить, что в этих двух статьях, как и в большинстве других, авторы уделают крайне мало внимания расширяемости системы по типам эффектов, и как следствие, по типам анализа, рассматривая простые `read/write` эффекты чтения-записи в перменную, которые отличаются очень простыми правилами комбинирования.

В качестве редкого примера эффектов, более интересных с концептуальной точки зрения, можно привести многопоточные условные эффекты из работы Flanagan, Freund, Lifshin «Types for atomicity: Static Checking and Inference for Java» (2008) [26]. Авторы ставят цель разработать систему эффектов, способную обнаруживать ошибки в программах на `Java`, связанные с многопоточностью. В связи с этим вводится класс эффектов, связанных с различными типами многопоточного взаимодействия. Наиболее интересно, что эти эффекты являются условными, т.е. выполняются не всегда, из-за того, что поведение многопоточных методов почти всегда зависит от владения той или иной блокировкой. Авторы сталкиваются с некоторыми проблемами, свойственными условным эффектам – в частности, экспоненциальным взрывов длины аннотаций при последовательном комбинировании эффектов (мы подробней будем говорить об этом в главе 2).

Следует отметить замечательную особенность эффектов – вычислительная сложность анализа зависит от количества аннотаций эффектов и их типов, но не от длины кода (разумеется, в отсутствии автоматического вывода эффектов). Таким образом, пользователь подобной системы «платит только за то, что использует»: работа с проектом без аннотаций будет очень похожа на то, как если бы системы эффектов не было вовсе. Более того, пользователь может захотеть проаннотировать только некоторую подсистему, либо же использовать ровно один простой тип эффектов (какие-нибудь проверяемые исключения) – в том и в другом случае, он может легко полу-

чить желаемый результат без значительной платы процессорным временем.

Взглянем теперь в сторону практических реализаций систем эффектов.

Реализацией идеи систем эффектов по оригинальной статье является язык программирования FX-87 [14]. К несчастью, этот язык на сегодняшний день уже не поддерживается и не развивается (чему он, вероятно, в значительной степени обязан в высшей степени нечитаемому синтаксису). Кроме того, данный язык является функциональным, что еще более уменьшает актуальность этой реализации для настоящего исследования.

Намного более практико-ориентированной, и потому более интересной реализацией является *Checker Framework* [5]. Это фреймворк, построенный на основе JSR-308 [7], и позволяющий организовывать небольшие системы эффектов на основе `Java`-аннотаций. Фреймворк предоставляет набор готовых аннотаций, равно как и возможность задавать собственные.

Checker Framework построен на понятии *checker*-ов – небольших модулей, отвечающих за решение одной конкретной задачи анализа (с чисто технической точки зрения, каждый такой модуль является процессором аннотаций). *Checker*-ы независимы друг от друга, и могут добавляться в систему по мере необходимости, что обеспечивает масштабируемость по типам анализа.

Полезным наблюдением, которое можно извлечь из принципа устройства *Checker Framework* является тот факт, что подавляющее большинство эффектов нетривиально взаимодействуют только с эффектами такого же типа и никак не влияют на все остальные - именно на этом основывается модульная структура с *checker*-ами. Это дает очень важное понимание того, насколько хорошо ведут себя системы эффектов при масштабировании – сложность добавления нового эффекта мало зависит от количества уже существующих эффектов (при условии, что он с ними не взаимодействует, что чаще всего действительно так). Это второе очень важное свойство систем эффектов, помимо независимости вычислительной сложности анализа от длины кода.

1.1.2. Контракты

Подходом к той же самой проблеме, но несколько с другой стороны, являются **контракты**.

Определение 1.2. Контракт – это формальная, точная и проверяемая спецификация программной единицы, описывающая ее взаимодействие с другими программными единицами [17]

Чаще всего в качестве программной единицы используется функция. Контракты для функций описываются предусловием, постусловием и инвариантом. Предусловие – это набор ограничений на окружение, которые функция ожидает видеть выполненными. Постусловие – это условия, которые функция обязуется выполнить, если было выполнено предусловие. Инвариант – это условия, которые функция обязуется выполнять всегда.

Внимательно вдумавшись в определение 1.2, можно заметить, что оно очень близко по смыслу к определению 2.3. Это и не удивительно, т.к. оба метода ставят своей целью явно специфицировать межпроцедурные взаимодействия. Отличие контрактов от эффектов заключается больше в синтаксисе записи и терминологии, нежели в концептуальных различиях. В достаточно мощной системе эффектов можно выразить понятие контракта, и обратно, с помощью контрактов и некоторого числа дополнительных конструкций можно записать эффекты функции. Разумеется, некоторые эффекты легче укладываются в концепцию контрактов, и наоборот.

Несмотря на схожесть контрактов и эффектов с точки зрения выразительности, между ними есть разница, если задуматься о чисто практических вопросах – а именно, о том, какого рода утверждения нужно будет чаще формализовывать в рамках системы анализа.

Системы эффектов часто являются интуитивно понятными, когда каждый эффект говорит о наличии либо отсутствии некоторого четко определенного и кратко описываемого свойства у функции – возможности бросить исключение, чистоты, и т.д. Само по себе свойство может быть довольно нетривиальным – например, если пытаться формализовать понятие чистоты не используя непосредственно термин «чистая функция», то придется

формализовать довольно длинную мысль: «на одним и том же наборе входных аргументов функция всегда возвращает одно и то же значение, и при этом выполнение функции никогда не вызывает наблюдаемых побочных эффектов». Однако важно, что будучи однажды введенным, это свойство записывается кратко и емко.

Контракты же удобны, если центральные понятия – предусловия и постусловия – будут часто использоваться. То же самое понятие чистоты функции никак не связано с предусловиями и постусловиями, и даже с понятием инварианта – это просто неотъемлемое свойство функции. Некоторым расширением синтаксиса контрактов, конечно, можно его выразить ровно также, как и в эффектов – но если в предметной области все утверждения похожи по своему устройству на чистоту, то смысла в контрактах остается не много.

Подводя итог, можно сказать, что в целом по своим свойствам контракты очень похожи на эффекты, а выбор конкретного подхода должен зависеть от предметной области и от того, какого рода утверждения будут чаще использоваться в итоговой системе.

1.1.3. Анализ потока данных

Анализ потока данных (англ. *data-flow analysis*) является методом статического анализа, возникшим в результате развития компиляторов, и, как следствие, потребности в эффективном и практичном способе доказательства *некоторых* свойств программы.

Отличие анализа потока данных от тех же языков спецификации заключается в том, что анализ потока данных не пытается доказать корректность программы в целом, а вместо этого сосредотачивается на некоторых, достаточно простых и полезных свойствах, например, константности переменных, типах и аттрибутах переменных, и т.д. [23]

Мы будем достаточно подробно обсуждать детали работы анализа потока данных чуть позже, здесь же ограничимся тем, что он анализирует непосредственно исходный код. В этом его основное отличие от систем эффектов и контрактов – анализ потока данных, с одной стороны, работает прозрачно для пользователя, не требуя от него никаких специальных аннотаций. С

другой же стороны, это приводит к некоторых техническим ограничениям. В частности, отсутствие исходного кода делает анализ невозможным, что приводит к некоторым проблемам при использовании в языках с раздельной компиляцией.

Кроме того, как несложно догадаться, вычислительная сложность анализа потока данных растет вместе с размером анализируемого кода [21]. В связи с этим, на практике наиболее распространен внутрипроцедурный анализ потока данных, т.к. он значительно проще в реализации, и вместе с тем все равно позволяет получать довольно полезные на практике утверждения [1].

В контексте данной работы интерес представляет межпроцедурный анализ потока данных. Одним из наиболее логичных и распространенных методов борьбы с возрастающей сложностью при межпроцедурном анализе является «сжатие» информации о подпрограммах [27, 2]. Суть достаточно проста – вместо того, чтобы пытаться «в лоб» проанализировать код всей программы, мы заменяем все подпрограммы на некоторые краткие описания ее эффектов и взаимодействий, релевантных для конкретного типа анализа. После этого, межпроцедурный анализ начинает напоминать внутрипроцедурный – при анализе конкретной подпрограммы, вместо кода использованных в ней других подпрограмм, используются полученные ранее краткие описания.

Как и следует ожидать, итоговые характеристики подобного подхода очень сильно зависят от того, как получаются эти «краткие описания». По этой причине мы не будем подробно анализировать возможные виды межпроцедурного анализа, основанные на этой идее, поскольку этот анализ будет довольно точно следовать анализу конкретного метода, использованного для извлечения кратких описаний.

1.1.4. Языки спецификаций

Определение 1.3. Языком спецификаций называется формальный язык, призванный описывать свойства и поведение систем на более высоком уровне абстракции, нежели языки программирования.

Как следует из определения, целью языков спецификации является как можно более подробное формальное описание систем. В связи с этим, большинство современных известных языков спецификации являются очень мощными системами, построенных вокруг фундаментальных понятий алгебры, математической логики, теории типов, теории категорий.

Например, язык спецификаций *Z-нотация* (англ. *Z-notation*) основывается на теории множеств, лямбда-исчислении и логики предикатов первого порядка [24]. Язык *CASL* поддерживает логику первого порядка, индукцию, частичные функции, наследование, и др. [4].

Благодаря мощному инструментарию, языки спецификации могут формализовывать довольно сложные и нетривиальные утверждения, связи и взаимодействия в системе. Следует ожидать, что в языках спецификации можно выразить и системы эффектов, и контракты, и это действительно так – язык *Larch* использует ключевые слова `require`, `ensure` для описания предусловий и постусловий [10]

В силу врожденной гибкости и выразительности, языки спецификаций обладают впечатляющей расширяемостью по типам анализа. Ценой же этому становится сложность. Подобные языки вводят значительное количество новых концепций и понятий, а также требуют от пользователя знания фундаментальных теорий, положенных в основу языка (чаще всего, алгебры и математической логики). Спецификации к программам могут быть длиннее и сложнее самих программ. В связи с этим, языки спецификаций в основном находят свое применение в тех областях разработки ПО, где недопустимы даже малейшие сбои и ошибки – например, при разработке аппаратов радиотерапии [8].

Таким образом, языки спецификаций – это отдельный, готовый инструмент, предназначенный в большей степени для формального доказательства корректности программ, нежели чем для улучшения качества статического анализа. Тем не менее, их выразительность, гибкость, а также в целом возможность воспользоваться готовым решением вроде CASL или Larch может иногда выглядеть довольно заманчиво.

1.2. Предметная область

До этого момента мы, в основном, говорили несколько абстрактно и неконкретно. Это и неудивительно, т.к. такие понятия, как «язык формальной спецификации» или «контракты» являются очень широкими, способными гибко подстраиваться под любую предметную область и ее требования.

Вместе с тем, продолжать дальнейший разговор без предъявления некоторых реальных проблем видится видится крайне малоосмысленным. Поэтому в данном разделе будет рассказана краткая вводная в язык `Kotlin` и несколько реальных проблем синтаксического анализа в нем, связанных с межпроцедурным анализом. Это позволит сформулировать несколько несложных требований к разрабатываемой системе, которые будут служить нам базовой проверкой ее адекватности реальным задачам.

1.2.1. Механизм умных приведений типов

Важным механизмом, понимание которого необходимо для дальнейшего разговоре о `Kotlin`, является механизм **умных приведений типов** (англ. *smartcasts*).

Очень часто, перед явным приведением типа в коде выполняется проверка на подтип (для того, чтобы такое приведение было безопасным). В `Java` это выглядит примерно следующим образом:

```
1  if (x instanceof string) {
2      string s = (string) x;
3      ... usage of s ...
4  }
```

Суть механизма умных приведений типов заключается в том, чтобы облегчить программисту жизнь и выполнить приведение к более частному типу автоматически там, где это безопасно.

Вот эквивалентный участок кода на `Kotlin`:

```
1  if (x is String) {
2      val len = x.length
3  }
```

Во второй строке произошло автоматические приведение типа, благодаря чему стал возможным доступ к полю `String.length`.

Особенно удобно это вместе с `when`-конструкцией, которая является более мощным аналогом `switch-case` из `Java`, в частности, позволяя использовать в `Kotlin` синтаксис, напоминающий сравнение с образцом (англ. `pattern matching`).

```
1  when (x) {
2      is String -> x.length   // x casted to String
3      is List<Int> -> x.size  // x casted to List
4      is Double -> x * 2.0    // x casted to Double
5  }
```

1.2.2. Проблемы механизма умных приведений типов

Основная проблема заключается в том, что анализ возможности приведения типа выполняется без учета межпроцедурных взаимодействий. Так, давайте рассмотрим функцию проверки на строковый тип:

```
1  fun isString(x: Any?): Boolean {
2      return x is String
3  }
```

Теперь если использовать эту функцию в качестве условия условного оператора, то умное приведение типа выполнено не будет:

```
1  if (isString(x)) {
2      val s = x as String    // explicit cast needed
3  }
```

Особенно актуальна эта проблема для работы с коллекциями с использованием stream-like API, предоставляемом в `Kotlin`. Так, существует метод `filter`, который оставляет в коллекции только те элементы, на которых переданный предикат вернул `true`. Этот метод довольно часто используется для того, чтобы оставить в коллекции только объекты определенного типа: `list.filter(x -> x is String)` оставит в коллекции `list` только строки.

Разумеется, в обоих описанных случаях компилятор не может просто так выполнить умное приведение типа – для этого ему нужно знать некоторый «контракт» вызываемой функции.

Для `isString` нужно знать, что «`isString(x)` возвращает `true` тогда и только тогда, когда `x` – `String`».

Для `filter` нужно знать, что он оставляет в коллекции только те объекты, на которых переданный предикат вернул `true`.

Другим распространенным случаем, когда умное приведение типов можно было бы сделать, но оно не делается, являются функции, которые могут завершиться с исключением.

Рассмотрим следующий пример:

```
1  assert(x is String)
2  val s = x as String    // explicit cast needed
```

Если мы будем считать, что `assert` бросает исключение всегда, когда его условие – `false`, тогда во второй строке автоматическое приведение типа может быть безопасно выполнено. Тем не менее, на данный момент компилятор `Kotlin` этого не делает, ровно по тем же причинам, что и в примерах выше – он не выполняет межпроцедурного анализа, и потому ему не известен контракт «assert».

1.2.3. Анализ инициализации переменных

`Kotlin` поддерживает отложенную инициализацию локальных переменных и будет отслеживать и предупреждать об использовании неинициализированных переменных:

```
1  val x: Int
2  // println(x)
3  x = 5
4  println(x)
```

Если раскомментировать строку 2, то компилятор вполне законно выдаст ошибку об использовании неинициализированной переменной.

Теперь рассмотрим функцию `run`, которая просто вызывает переданную ей лямбду:

```
1  fun run(block: () -> Unit): Unit {
2      block()
3  }
```

Если теперь выполнить отложенную инициализацию переменной внутри лямбды, переданной внутрь `run`, то компилятор уже не сможет доказать, что переменная была корректно проинициализированна:

```
1  val x: Int
2  run ({ () -> x = 5 })
3  println(x)
```

Компилятор отвергает подобный код, ошибочно сообщая об использовании неинициализированной переменной в строке 3. Причина этого примерно такая же, как и во всех предыдущих примерах: компилятор не знает контракта функции `run`, и потому не знает, что лямбда `{ () -> x = 5 }` будет гарантированно вызвана.

Этот пример умышленно утрирован для простоты объяснения. На самом деле, это вполне актуальная и серьезная проблема, т.к. `Kotlin` придерживается философии введения как можно меньшего количества ключевых слов. Это возможно из-за существования лямбд и функций высшего порядка, а также из-за синтаксического сахара, позволяющего передавать лямбду

в виде блока, если она передается последней в списке параметров. Именно так реализован аналог ключевого слова `synchronized` из `Java`: в `Kotlin` это обычная функция, определенная примерно следующим образом (детали взятия и освобождения блокировки опущены для ясности).

```
1  fun synchronized(block: () -> Unit): Unit {
2      ... take lock ...
3      block()
4      ... release lock ...
5  }
```

В пользовательском коде использование этой функции с учетом описанного выше синтаксического сахара выглядит следующим образом:

```
1  val x: Int
2  synchornized {
3      x = 5
4  }
5  println(x)
```

Точно также, как в примере с `run`, компилятор заявляет о том, что в строке 5 переменная `x` не инициализирована. Это уже более серьезная проблема, т.к. здесь нельзя обойтись парой избыточных символов, как это было в примере с умными приведениями типа. q Такой случай использования является вполне жизненным и распространенным. Например, абсолютно также реализована идиома `try-with-resources` из `Java`, различные функции для работы с корутинами, и т.д.

Для того, чтобы решить это проблему, необходимо некоторым образом донести до компилятора контракт всех таких функций: что они вызывают переданную им лямбду некоторое статически детерминированное число раз.

1.3. Актуальность, цель и задачи работы

Подведем краткий итог проблем, которые мы пронаблюдали в предметной области:

- Корнем всех перечисленных выше проблем являются неучтенные межпроцедурные взаимодействия
- Для того, чтобы корректно дополнить анализ, необходимо извлечь некоторые факты и утверждения о том, как именно работают функции (связь входного и выходного значения, как в `isString`, или же информация о поведении функции по отношению к переданным аргументам, как в `run`)
- Следует ожидать, что контракты могут становиться весьма нетривиальными, равно как и то, что даже достаточно простые контракты может оказаться непросто выводить в автоматическом режиме (например, функция может действительно вызывать лямбду ровно один раз, но при этом делать еще множество других нетривиальных вещей, тем самым затрудняя анализ).

На основе этих утверждений, можно сформулировать ряд ожидаемых требований:

1. Система должна быть способна описывать межпроцедурные взаимодействия с условиями (например, связь между аргументами функции и возвращаемым значением).
2. Система должна поддерживать явные пользовательские аннотации.
3. Система должна быть удобной для интеграции с существующими решениями (компиляторами, IDE).
4. Вычислительная сложность анализа должна позволять работу с крупными проектами (от сотен тысяч строк кода).
5. Система должна по возможности легко расширяться по типам анализа.

Вспомним рассмотренные нами подходы. Их соответствие означенным критериям сведено в таблицу 2.

	Условные зависимости	Явные аннотации	Удобство интеграции	Алгоритм. сложность	Расширяемость
Анализ потока данных	+	-	+	-	+
Языки спецификаций	+	+	-	+	+
Эффекты	+/-	+	+	+	+/-
Контракты	+	+	+	+	+/-

Таблица 2: Сравнение различных подходов к межпроцедурному анализу

Прокомментируем некоторые моменты.

- Языки спецификации очень неудобны для интеграции в существующие решения. В особенности это ощущается, если между существующим решением и языком спецификации поток информации двунаправленный – добавить в компилятор логику обращения за информацией в язык спецификаций относительно несложно, но вот сделать обратное может быть или очень нетривиально, или зачастую вовсе невозможно.

- Символ «+/-» следует читать как «система потенциально удовлетворяет критерию, но вопрос реализации изучен слабо»

Можно видеть, что ни одно из существующих решений не удовлетворяет всем требованиям. Однако, эффекты и контракты видятся как наилучшие подходы к решению поставленных проблем. Для того, чтобы выбрать между этими двумя подходами, нужно вспомнить, что при анализе мы отмечали их большую схожесть, и сделали вывод, что выбор необходимо делать исходя из предметной области.

Внимательное рассмотрение проблем, поставленных в предметной области, дает понять, что в данном конкретном случае концепция предусловий-постусловий не является востребованной, и зачастую не используется, по-

этому было принято решение взять за основу системы эффектов. Таким образом, можно сформулировать цель и задачи:

Цель. Разработать систему эффектов, выполняющую анализ кода посредством использования информации о поведении вызываемых функций, и реализовать ее прототип в компиляторе языка `Kotlin`

Задачи:

1. Выполнить анализ существующих решений
2. Разработать систему эффектов
3. Реализовать разработанную систему в компиляторе `Kotlin`.
4. Провести анализ полученного решения, выявить его достоинтсва и недостатки.

2. Устройство систем условных эффектов

2.1. Основные понятия

2.1.1. Понятие условного эффекта

Центральным определением в данной работе, является, разумеется, **эффект**. Однако несмотря на то, что это определение, судя по всему, было введено еще на заре развития программирования (так, ранние работы по аксиоматизации программирования уже ссылаются на этот термин без отдельного его введения [11, 22]), общепринятой формулировки за все это время не появилось.

В классических источниках, под эффектом чаще всего понимают «некоторое видимое изменение в окружении» [15], или даже еще более конкретно «изменение в памяти программы» [12]. Некоторые исследователи и вовсе ограничивают это определение до «чтения или записи в изменяемое состояние программы» [9]. Это можно резюмировать следующим образом:

Определение 2.1. Эффектом называется некоторое изменение, производимое подпрограммой в состоянии вычислителя (кроме возвращения подпрограммой значения).

Другие же авторы употребляют более широкую трактовку «эффекта» [19]:

Определение 2.2. Эффектом является описание действий, происходящих в ходе выполнения подпрограммы.

Разумеется, формулировка 2.2 является слишком широкой – вплоть до того, что под нее подходит непосредственно исходный код тела функции. С другой же стороны, формулировка 2.1 является нежелательно узкой в контексте данной работы. Поясним это на примере:

```
1 fun always42(): Int {
2     return 42
3 }
```

Мы бы хотели говорить, что данная функция имеет эффект «Всегда возвращает число 42». Однако это действие не подходит под определение эффекта 2.1. Мы могли бы отказаться от специального случая для возвращаемых значений в этой формулировке, но в дальнейшем мы встретим некоторые утверждения, которые мы тоже хотели бы называть эффектами, но которые не описывают вообще никакого изменения в состоянии вычислителя.

Поэтому нам понадобится определение, чуть более слабое, чем определение 2.1, но при этом не являющееся чересчур расплывчатым, как 2.2. Мы сформулируем его следующим образом:

Определение 2.3. Эффект – это некоторая информация об окружении, получаемая при выполнении подпрограммы.

Т.к. это определение рассматривает только окружение, то сразу отпадают все слишком широкие его интерпретации. В частности, все, что подпрограмма делает со своими локальными переменными, не подходит под это определение – что очень удобно, т.к. изменения в локальных переменных нас никоим образом не интересуют.

С другой стороны, это определение включает в себя определение 2.1, т.к. «изменение в состоянии», несомненно, является «информацией об окружении».

Наконец, как мы увидим чуть позже, под это определение подходят и довольно нестандартные действия, которые нам будет удобно считать эффектами во имя общности подхода.

Однако, даже такое, слегка расширенное понятие эффекта недостаточно для того, чтобы формализовать устройство многих функций.

Для примера рассмотрим следующую функцию, которая является тривиальной оберткой над проверкой переменной на принадлежность строковому типу:

```
1 fun isString(x: Any?): Boolean {
2     return (x is String)
3 }
```

Мы бы хотели отразить тот факт, что функция возвращает `true` тогда и только тогда, когда `x` является `String`. Для этого формализуем понятие условного эффекта:

Определение 2.4. Утверждением будем называть пару (c, e), где e – эффект, а c – некоторый набор условий. Семантика этой пары такова, что эффект e имеет место тогда, когда c выполняется.

В дальнейшем мы будем иногда называть условие c **посылкой**, и записывать эту пару следующим образом: $c \to e$.

Отметим, что обычные эффекты естественным образом выражаются через условные, если в качестве посылки использовать выражение, которое всегда выполняется (немного забегая вперед, скажем, что в качестве такого выражения мы будем использовать `true`, т.е. логическую истину).

2.1.2. Понятие схемы эффектов

Итак, у нас есть понятие «утверждение», описывающее условный эффект. Однако этого еще недостаточно для наших целей. Достаточно взглянуть даже на очень простой пример с функцией `isString(x)`, выдающей результат проверки аргумента на принадлежность строковому типу, чтобы понять, что утверждения часто используются в группах. Так, чтобы описать поведение функции `isString(x)`, на самом деле необходимо два утверждения:

- Если `x is String` верно, то функция возвращает `true`
- Если `x is String` неверно, то функция возвращает `false`

Поэтому в рамках данной работы было введено понятие **схемы эффектов**, формализующее данную идею:

Определение 2.5. Схемой эффектов называется набор *независимых* утверждений, описывающих условные эффекты некоторого участка кода.

Обратим внимание, что определение говорит, о *независимости* утверждений в схеме, т.е. исполнение или неисполнение некоторого конкретного

утверждения не влияет на другие. Важно понимать, что при этом может быть верным буквально *любое* подмножество утверждений, в том числе и пустое.

Также заметим, что используя независимые утверждения, легко выразить случаи, когда выполнение одного условия влечет сразу несколько эффектов – для этого достаточно выписать несколько утверждений с одним и тем же условием и разными эффектами.

Важно понимать, что схема эффектов не обязана описывать все эффекты вычисления, и в этом смысле она является не полной спецификацией. С другой стороны, если уж какой-то эффект был упомянут, должны быть полностью описаны все условия, которые его вызывают – т.е., относительно конкретного эффекта схема обязана быть полной.

Это вполне логично, если вспомнить *Checker Framework* и его архитектуру чекеров, каждый из которых представляет отдельно взятую систему эффектов. Функция не обязана перечислять свое поведение относительно каждого из существующих чекеров, но если уж она взялась специфицировать эффекты из некоторого чекера, эта спецификация должна быть корректна и полна – иначе этот самый чекер не сможет выдать никакого адекватного анализа.

Таким образом, все эти оговорки нужны для того, чтобы, с одной стороны, не заставлять специфицировать абсолютно все поведение функции (что может быть довольно сложно), но, с другой стороны, чтобы имеющаяся спецификация была полезной.

2.1.3. Краткая грамматика языка описания эффектов

Теперь, определившись с основными понятиями, мы можем ввести грамматику для записи схем эффектов и утверждений. Для описания здесь будет использоваться синтаксис, близкий к EBNF (расширенной нормальной форме Бэкуса-Науэра). Мы будем придерживаться следующих соглашений:

- Терминалы начинаются с большой буквы, например: `SimpleTerminal`
- Нетерминалы начинаются с маленькой буквы: `nonterminalSymbol`

- Каждая продукция начинается с двоеточия «: » и заканчивается точкой с запятой «;»
- Операторы «|», «*», «+», «?» несут стандартный смысл альтернативы, итерации (ноль или более), итерации (один или более) или опции (один или менее) соответственно.
- Кроме того, мы будем писать $\alpha\{\beta\}$, чтобы обозначить непустой список из символов α, разделенных символом β.

Мы приводим здесь неформальную грамматику, опуская технические детали вроде леворекурсивных правил, точных определений литералов, приоритетов и т.д. Также, в данной работе мы иногда будем немного вольно обращаться с некоторыми элементами синтаксиса во имя ясности записи (в частности, мы будем позволять иногда опускать скобки там, где это не может привести к двусмысленности).

Полная грамматика в синтаксисе `ANTLR` приведена в приложении А.

```
1  effectSchema : statement{ EOL } ;
2
3  statement : expression '->' effect{ ',' } ;
4
5  expression : operator | Constant | Variable ;
6
7  operator : isOperator | andOperator | orOperator | equalOperator ;
8
9  effect : returnsEffect | throwsEffect | callsEffect ;
10
11 // Операторы
12 isOperator     : Variable 'is' Type ;
13 andOperator    : expression '&&' expression ;
14 orOperator     : expression '||' expression ;
15 equalOperator  : expression '==' expression ;
16
```

```
17 // Эффекты
18 returnsEffect : 'Returns' '(' (expression | Wildcard)  ')' ;
19 throwsEffect  : 'Throws'  '(' (Type       | Wildcard)  ')' ;
20 callsEffect   : 'Calls'   '(' (Variable   | Wildcard) ',' Constant ')' ;
21
22 // Терминалы
23 Constant : <численная либо строковая константа> ;
24 Variable : <корректный идентификатор> ;
25 Type     : <корректный идентификатор> ;
26 Wildcard : '???' ;
27 EOL      : <разрыв строки> ;
```

Семантика операторов:

- `Is-оператор` – выдает результат проверки переменной на принадлежность типу, подробнее см. в [25].
- `And-оператор` – соответствует логической конъюнкции.
- `Or-оператор` – соответствует логической дизъюнкции.
- `Equal-оператор` – соответствует проверке на равенство (equality) в `Kotlin`, подробнее см. в [6].

Семантика эффектов:

- `Returns(x)` говорит, что данный участок кода при исполнении возвращает значение `x`.
- `Throws(e)` говорит, что в результате исполнения данного участка кода генерируется исключение `e`.
- `Calls(f, c)` говорит, что в результате исполнения данного участка кода `c` раз будет вызвана функция `f`.

Кроме того, нам понадобилось ввести особый символ подстановки «???», означающий неизвестность. Он необходим для того, чтобы можно было записать утверждения вроде: `Returns ???` («участок кода завершается успешно, но конкретное возвращаемое значение неизвестно»), или `Throws ???` («участок кода завершается неуспешно»). Особенно большое значение наличие таких конструкций будет иметь при извлечении полезной информации из системы эффектов, о чем мы будем говорить в главе 3.

Заметим, что приведенная выше грамматика естественным образом индуцирует дерево, которое мы будем называть **деревом схемы эффектов**, или просто **дерево схемы**. Фактически, если рассматривать схему как некоторое выражение в грамматике, определенной выше, то дерево этой схемы соответствует дереву разбора этого выражения, из которого удалены все узлы-литералы (такие как разделители, скобки и т.д.).

2.1.4. Изоморфизм схем

В будущем мы будем определять некоторые трансформации над схемами эффектов. Большинство этих трансформаций в каком-то смысле будут изменять лишь структуру схему, оставляя заложенную в нее информацию неизменной. Опять же, мы могли бы дать полностью формальное определение этому понятию, но это потребовало бы непропорционально больших усилий, поэтому ограничимся интуитивным:

Определение 2.6. Будем говорить, что схемы A и B являются **изоморфными**, если они описывают один и тот же набор условных эффектов. Трансформацию, которая переводит схему A в схему B будем называть **изоморфизмом схем**.

Интуитивно, мы можем спокойно заменить схему изоморфной, и это никак не повлияет на результаты анализа кода, полученные с помощью этой схемы. Например, следующие две схемы являются изоморфными, хотя с чисто синтаксической точки зрения в них записаны разные утверждения:

```
schema A:
[ x == true && y is Int → Returns 1     ]
[ x == false || y !is Int → Returns 0   ]
```

schema B:

$$\left[\begin{array}{l} \texttt{!(x == false || y !is Int)} \rightarrow \texttt{Returns 1} \\ \texttt{!(x == true \&\& y is Int)} \rightarrow \texttt{Returns 0} \end{array}\right]$$

2.1.5. Примеры

Теперь мы ввели все необходимые понятия для того, чтобы научиться аннотировать некоторые функции с не слишком сложными контрактами (в частности, обсуждение уточнений типов в коллекциях мы на некоторое время отложим). В данном разделе мы приведем несколько примеров для того, чтобы наглядно продемонстрировать работу с описанным синтаксисом и терминами.

Начнем с рассмотрения уже использовавшейся нами не раз функции `isString(x)`:

```
1  fun isString(x: Any?): Boolean {
2      return (x is String)
3  }
```

Ей соответствует следующая схема:

schema isString:

$$\left[\begin{array}{l} \texttt{x is String} \rightarrow \texttt{Returns true} \\ \texttt{x !is String} \rightarrow \texttt{Returns false} \end{array}\right]$$

Как мы видим, эта схема в точности передает контракт `isString(x)`: функция возвращает `true` если переданный аргумент является подтипом `String` и возвращает `false` в противном случае.

Заметим, что в силу того, что мы используем независимые утверждения, схема эффектов не отражает явным образом тот факт, что эти два утверждения описывают *все* возможные результаты выполнения данной функции. Неявно это выражено тем, что для любого объекта `x` верно либо `x is String`, либо `x !is String`.

Другой пример, который мы хотели бы выразить в системе эффектов, это функция `assert`. Мы уже упоминали это соглашение, и здесь мы повторим его: мы будем считать, что `assert` всегда генерирует исключение, если

переданный аргумент равен `false`. На практике это верно, например, для различных тестовых фреймворков, типа *JUnit*.

```
1  fun assert(cond: Boolean): Unit {
2      if (!cond) {
3          throw AssertionError("Assertion Failed")
4      }
5  }
```

Этой функции соответствует следующая схема:

schema assert:

$$\left[\begin{array}{l} \texttt{cond == true} \rightarrow \texttt{Returns(Unit)} \\ \texttt{cond == false} \rightarrow \texttt{Throws AssertionError} \end{array}\right]$$

Эта запись отражает контракт функции `assert`: она завершается успешно тогда и только тогда, когда аргумент равен `true`. Опять же, факт того, что описание в схеме исчерпывающее, выражен неявно – булева переменная либо истинна, либо ложна.

Наконец, рассмотрим функции типа `run`, которые в ходе своей работы вызывают некоторую другую детерминированное число раз:

```
1  fun run(block: () -> Unit): Unit {
2      block()
3  }
```

Этой функции соответствует схема:

schema run:

$$\left[\begin{array}{l} \texttt{true} \rightarrow \texttt{Calls(block, 1)} \end{array}\right]$$

Данная запись отражает то, что функция `run` *всегда* (посылка `true` всегда истинна) вызывает переданный аргумент `block` ровно один раз.

Отметим, что в данном случае схема специфицирует лишь *часть* контракта функции. В частности, она не отражает тот факт, что функция завершается успешно тогда и только тогда, когда успешно завершается вызов `block()`.

2.2. Использование схем эффектов

2.2.1. Подстановка аргументов

Мы научились описывать базовые схемы эффектов для функции. При этом мы не заостряли внимание на том, что все эти схемы используют *параметры* функции. С точки зрения формализма, эти схемы не имеют смысла при вызове функции с конкретными аргументами. Действительно, пусть у нас есть схема для все той же функции `isString`:

```
schema isString(x):
[ x is String → Returns true
  x !is String → Returns false ]
```

И пусть мы вызываем эту функцию в коде:

```
1  val someValue: Any?
2  <initialize x somehow>
3
4  if (isString(someValue)) {
5      println("someValue is String!")
6  }
```

Чисто формально, в строке 4 мы хотели бы получить схему, в которой используется `someValue`, а не `x`:

```
schema isString(someValue):
[ someValue is String → Returns true
  someValue !is String → Returns false ]
```

Т.е. мы должны некоторым образом связать формальные параметры (в данном случае `x`) и аргументы, использованные при вызове (в данном случае `someValue`). Это довольно известный процесс **подстановки**, часто рассматриваемый, например, при описании лямбда-исчисления.

Мы не будем здесь выписывать всю формалистику, связанную с этой операцией, поскольку она, во-первых, абсолютно схожа с процессом подстановки в лямбда-исчислении, а во-вторых, она потребует введения нескольких классических понятий (например, альфа-эквивалентности), совершенно излишних в контексте данной работы.

В связи с этим, мы приводим здесь лишь интуитивное описание процесса подстановки, отсылая за подробностями в любой хрестоматийный труд по теории языков программирования, например, в [20].

Определение 2.7. Пусть схема S записана для объявления некоторой функции с формальным параметром `x`. Будем говорить, что «S **абстрагирована** по `x`», а `x` – **связана** в S.

Тогда **подстановкой выражения** t **вместо переменной** `x` **в схему** S будем называть схему $S[x \to t]$, которая в точности равна схеме S за исключением того, что любое вхождение `x` в ней заменяется на выражение t.

Классически, при определении подстановки мы сталкиваемся с проблемой захвата переменной – если схема S абстрагирована по `x` и при этом содержит в себе другую схему Q, которая также абстрагирована по `x`, то подставлять t вместо `x` в Q некорректно.

Обычно эта проблема решается с помощью простого наблюдения: имена связанных переменных не важны, и могут быть выбраны произвольно. Тогда мы можем переименовать все переменные так, чтобы все имена были уникальны. Этого весьма неформального утверждения будет достаточно для наших целей.

2.2.2. Сглаживание схем эффектов

Теперь мы умеем использовать схемы эффектов в простых вызовах. Однако этого пока что недостаточно для практического использования: в частности, вложенные вызовы порождают вложенные схемы, работа с которыми неудобна:

```
1  val x: Any?
2  ...
3  <initialize x somehow>
4  ...
5  assert(isString(y))
```

При построении схемы для данного вызова необходимо выполнить две подстановки: сначала $isString[x \to y]$ (результат обозначим как $isString'$), потом $assert[cond \to isString']$. Более подробно:

```
schema isString(y):
⎡ y is String -> Returns true  ⎤
⎣ y !is String -> Returns false ⎦
```

```
schema assert(isString(y)):
⎡ schema isString(y):                                                    ⎤
⎢ ⎡ y is String -> Returns true  ⎤                                       ⎥
⎢ ⎣ y !is String -> Returns false ⎦ == true → Returns Unit               ⎥
⎢ schema isString(y):                                                    ⎥
⎢ ⎡ y is String -> Returns true  ⎤                  Throws               ⎥
⎣ ⎣ y !is String -> Returns false ⎦ == false →      AssertionError       ⎦
```

При этом хотелось бы, чтобы схема для вложенного вызова отражала итоговую его семантику, а именно: вызов завершается успешно тогда и только тогда, когда `y` является `String`. На языке схем эффектов это будет выглядеть так:

```
schema assert(isString(y):
⎡ y is String → Returns Unit          ⎤
⎣ y !is String → Throws AssertionError ⎦
```

Для этого нам нужно ввести операцию **сглаживания** схем:

Определение 2.8. Пусть S – некоторая схема эффектов. Тогда результатом **многошагового сглаживания** этой схемы будем называть схему S_{flat}, такую, что она не содержит вложенных схем эффектов и при этом изоморфна схеме S. Мы иногда будем называть S_{flat} **плоской**.

Мы хотели бы перейти к задаче **одношагового сглаживания**: требуется сгладить узел, такой, что некоторые его аргументы являются схемами эффектов, но при этом все эти схемы эффектов сами по себе уже являются плоскими. Это помогло бы нам тем, что имея операцию одношагового сглаживания, легко построить операцию многошагового сглаживания с использованием рекурсии:

```
1  fun ManyStepFlatten(root) {
2      for (i in 1 to |root.childs|) {
3          v_i  ← root.childs[i]
4          v'_i ← ManyStepFlatten(v_i)
5      }
6      return OneStepFlatten(root, {v_1, v_2, ..., v_|root.childs|})
7  }
```

Действительно, рано или поздно $ManyStepFlatten$ доберется до листьев иерархии, тем самым вырождаясь в вызов $OneStepFlatten$. Листья, при этом, разумеется, и так плоские, поэтому требование для $OneStepFlatten$ выполнено. При обратном ходе рекурсии требование плоскости аргументов $OneStepFlatten$ также будет выполняться согласно определению $ManyStepFlatten$.

Осталось определить $OneStepFlatten$. Внимательное рассмотрение грамматики позволяет заметить, для этого достаточно определить $OneStepFlatten$ для трех случаев (одношаговое сглаживание плоских узлов выполняется тривиально):

1. Сглаживание оператора, хотя бы один из аргументов которого – схема эффектов
2. Сглаживание утверждения, посылка или заключение которого – схема эффектов
3. Сглаживание схемы эффектов, у которой хотя бы одно из утверждений является схемой эффектов.

Сразу заметим, что одношаговое сглаживание схемы эффектов также определяется достаточно просто и соответствует объединению утверждений всех аргументов.

Конечные версии этих алгоритмов, пригодных для использования на практике являются достаточно сложными и громоздкими. Поэтому мы сделаем следующие несколько предположений:

- Все операторы бинарные

- При сглаживании оператора будем считать, что все его аргументы являются схемами эффектов (а не только некоторые)
- Все вычисления являются тотальными

Мы начнем с достаточно простого алгоритма, работающего только при выполнении этих предположений. Затем мы будем постепенно усложнять его, показывая, как можно отказаться от каждого из этих предположений.

2.2.3. Сглаживание операторов. Базовый алгоритм

Пассмотрим некоторый бинарный оператор α. Пусть его аргументами являются два выражения A, B, с соответствующими схемами:

`schema A:`

$$\left[\begin{array}{l} c_1^A \to e_1^A \\ c_2^A \to e_2^A \\ \dots \\ c_n^A \to e_n^A \end{array}\right]$$

`schema B:`

$$\left[\begin{array}{l} c_1^B \to e_1^B \\ c_2^B \to e_2^B \\ \dots \\ c_m^B \to e_m^B \end{array}\right]$$

Необходимо построить схему эффектов, соответствующую вычислению $\alpha(A, B)$.

Сначала мы просто берем декартово произведение множеств утверждений схем A и B, получая множество пар утверждений вида:

$$A \times B = \left\{(c_i^A \to e_i^A, c_j^B \to e_j^B) | i \in 1 \dots n, j \in 1 \dots m\right\}$$

Можно видеть, что это множество пар содержит в себе *практически* все возможные пары эффектов, которые могли быть сгенерированы при вычислении аргументов оператора. Более того, ко всем этим парам эффектов прикреплены условия, и для того, чтобы *пара* эффектов выстрелила, необходимо, чтобы были выполнены *оба соответствующих* условия.

Можно также думать про эти пары как про возможные пути потока управления – сначала управление заходит в аргумент A, выполняется одно из условий c_i^A, выстреливают эффекты e_i^A, затем выполняется одно из условий c_j^B и выстреливают соответствующие эффекты e_j^B. Таким образом, на этом пути вычисления верны условия c_i^A и c_j^B и генерируются эффекты e_i^A, e_j^B.

Рассуждения выше должны были подвести к следующему преобразованию, которое склеивает пару утверждений в одно:

$$(c_i^A \to e_i^A, c_j^B \to e_j^B) \Rightarrow c_i^A \wedge c_j^B \to e_i^A, e_j^B$$

Однако здесь пока что еще нигде не появился оператор α. Вообще говоря, мы никак не ограничивали то, что оператор может сделать с аргументами, поэтому в целом мы должны сказать, что затем это утверждение подвергается преобразованию α, определяемому самим оператором:

$$\alpha(c_i^A \wedge c_j^B \to e_i^A, e_j^B)$$

Здесь мы могли бы и закончить, сказав, что результат этого преобразования может быть абсолютно произвольным, с любым изменением структуры утверждения, вплоть до полного его удаления.

Тем не менее, практика показывает, что большинство операторов устроены достаточно просто и такая общность является излишней. В частности, все операторы, которые будут рассмотрены в этой работе, действуют только и исключительно на заключения, оставляя посылки нетронутыми.

В этом есть некоторая логика, поскольку мы здесь интересуемся наблюдаемыми эффектами при вычислении оператора α. Запись $c_i^A \wedge c_j^B \to e_i^A, e_j^B$ говорит нам, что аргументы этого оператора при выполнении условий $c_i^A \wedge c_j^B$ влекут эффекты e_i^A, e_j^B. Но это еще не означает, что мы обязательно пронаблюдаем эти эффекты, т.к. между ними и наблюдателем еще стоит оператор α. Очевидно, что этот оператор может повлиять на наблюдаемые эффекты (например, обработав летящее исключение). Но что более интересно, из такого объяснения должно стать ясно, что оператор не должен влиять на условия $c_i^A \wedge c_j^B$, потому что это свойство *аргументов* оператора, но не его

самого и даже не вычисления оператора на этих аргументах.

Таким образом, мы определим преобразование оператора более точно:

$$\alpha(c_i^A \wedge c_j^B \to e_i^A, e_j^B) \equiv c_i^A \wedge c_j^B \to \alpha(e_i^A, e_j^B)$$

Что дает нам следующую схему эффектов:

`schema` α`(A, B):`

$$\left[\begin{array}{l} c_1^A \wedge c_1^B \to \alpha(e_1^A, e_1^B) \\ c_1^A \wedge c_2^B \to \alpha(e_1^A, e_2^B) \\ \ldots \\ c_1^A \wedge c_m^B \to \alpha(e_1^A, e_m^B) \\ c_2^A \wedge c_1^B \to \alpha(e_2^A, e_1^B) \\ c_2^A \wedge c_2^B \to \alpha(e_2^A, e_2^B) \\ \ldots \\ c_n^A \wedge c_m^B \to \alpha(e_n^A, e_m^B) \end{array}\right]$$

Заметим, что данный алгоритм естественно обобщается на операторы другой арности – для оператора арности r и аргументов $A_1, A_2, \ldots, A_r$ следует просто взять за основу r-местное декартово произведение $A_1 \times A_2 \times \ldots \times A_r$. Таким образом, мы уже отказались от предположения, что все операторы бинарные.

Остается еще один небольшой нюанс. Мы сказали, что декартово произведение $A \times B$ содержит «почти» все эффекты вычисления аргументов A и B. Какие же эффекты были упущены? Для ответа на этот вопрос вспомним, что возможны такие корректные состояния контекста, что ни одно из утверждений схемы не выполняется.

Перенося это на наш конкретный случай, это означает, что могут существовать такие состояния контекста, что ни одно из c_i^A не выполняется, но выполняются некоторые c_j^B. Отсюда мы бы могли наивно заключить, что вычисление аргументов оператора влечет соответствующие эффекты e_j^B, и записать $c_j^B \to e_j^B$. Однако это было бы, конечно, неверно. Действительно, вычисление аргумента A могло скрывать некоторые другие эффекты, и выполнение условия c_j^B влечет эффекты e_j^B *и еще какие-то неизвестные*, в то время как чуть ранее мы договорились, что запись $c_j^B \to e_j^B$ говорит о том,

что выполнение условия c_j^B влечет *только* эффекты e_j^B.

Потенциально, мы могли бы все-таки попытаться побороться за эти эффекты, но для этого потребуется введение новых понятий, либо расширение старых. По этой причине мы оставим попытки учесть такие эффекты. Схема, построенная по описанию выше, конечно, по-прежнему будет консервативным приближением динамического поведения соответствующего участка кода.

2.2.4. Приведение к схеме

Покажем, как отказаться от предположения, что все аргументы операторов являются схемами эффектов. Для этого вспомним еще раз рекурсивный алгоритм сглаживания:

```
1  fun ManyStepFlatten(root) {
2      for (i in 1 to |root.childs|) {
3          v_i  ← root.childs[i]
4          v'_i ← ManyStepFlatten(v_i)
5      }
6      return OneStepFlatten(root, {v_1, v_2, ..., v_|root.childs|})
7  }
```

Достаточно договориться, что *OneStepFlatten* всегда возвращает схему эффектов, и тогда предположение о том, что при сглаживании оператора все его аргументы являются схемами, будет выполнено. Действительно, перед сглаживанием оператора на всех его аргументах (детях в дереве схемы) вызывается *ManyStepFlatten*, который после рекурсивных вызовов возвращает результат *OneStepFlatten*.

В свою очередь, *OneStepFlatten* и так возвращает схемы эффектов при сглаживании операторов. Таким образом, остается лишь немного изменить операцию сглаживания на константах и переменных. Ранее мы говорили, что *OneStepFlatten* ничего не делает с такими узлами, т.к. они и так являются плоскими. Теперь же $OneStepFlatten(v)$, где v – это константа или переменная, будет возвращать следующую схему:

`schema OneStepFlatten(v):`

$$\left[\begin{array}{l} \texttt{true} \to \texttt{Returns(v)} \end{array}\right]$$

Действительно, с точки зрения эффектов константа или переменная v ничем не отличается от константной функции, всегда возвращающей v, схема которой и записана выше.

Таким образом, $OneStepFlatten$ всегда возвращает схему эффектов, следовательно, всегда возвращает схему эффектов $ManyStepFlatten$, следовательно, требование о том, что все аргументы оператора при сглаживании являются схемами, будет выполнено.

2.2.5. Сглаживание в присутствии частичных вычислений

У нас осталось одно предположение, которое не выполняется на практике – а именно, то, что все вычисления являются тотальными, т.е. всегда завершаются успешно. Отказаться от него будет уже не так просто, как от предыдущих двух, потому что частичные вычисления вынуждают намного более аккуратно обращаться с эффектами и учитывать, что аргументы оператора вычисляются последовательно.

Итак, рассмотрим бинарный оператор α, аргументами которого являются два выражения A, B, с соответствующими схемами:

`schema A:`

$$\left[\begin{array}{l} c_1^A \to e_1^A \\ c_2^A \to e_2^A \\ \dots \\ c_n^A \to e_n^A \end{array}\right]$$

`schema B:`

$$\left[\begin{array}{l} c_1^B \to e_1^B \\ c_2^B \to e_2^B \\ \dots \\ c_m^B \to e_m^B \end{array}\right]$$

Необходимо построить схему эффектов, соответствующую вычислению $\alpha(A, B)$, учитывая то, что вычисления могут обрываться, а аргумент A вычисляется раньше аргумента B.

Сначала поймем, почему нас не устраивает старый алгоритм. Пусть где-нибудь в схеме A есть утверждение $c_i^A \to$ `Throws Exception, Calls(f, 1)`, а где-нибудь в схеме B есть утверждение $c_j^B \to$ `Calls(f, 2)`. Тогда старый алгоритм выдаст схему $\alpha(A, B)$, которая будет содержать утверждение $c_i^A \wedge c_j^B \to \alpha$(`Throws Exception, Calls (f, 1), Calls (f, 2)`).

Нам же хотелось бы на выходе получить утверждение $c_i^A \wedge c_j^B \to$ `Throws Exception, Calls (f, 1)` – обратите внимание, что эффект `Calls(f, 2)` здесь не появляется, т.к. он порождается вычислением аргумента B, которое не произошло из-за того, что вычисление аргумента A сгенерировало ошибку. К сожалению, как бы ни было устроено преобразование α, оно не сможет отделить корректные эффекты от некорректных – на этом шаге уже потеряна вся информация, откуда пришел тот или иной эффект или условие.

Поэтому нам придется отступить на шаг назад. Вспомним, как было получено $\alpha(A, B)$. Сначала формировались все возможные пары утверждений $A \times B$:

$$A \times B = \left\{(c_i^A \to e_i^A, c_j^B \to e_j^B) | i \in 1 \dots n, j \in 1 \dots m\right\}$$

Затем каждая пара преобразовывалась в одно утверждение:

$$(c_i^A \to e_i^A, c_j^B \to e_j^B) \Rightarrow c_i^A \wedge c_j^B \to e_i^A, e_j^B$$

Именно на этом шаге была внесена непоправимая ошибка. Вводя данное преобразование, мы приводили аналогию с путями исполнения – поскольку ранее мы считали, что все вычисления тотальны, то было корректно считать, что после $c_i^A \to e_i^A$ управление передается в $c_j^B \to e_j^B$. Сейчас это не так, и необходимо аккуратнее учесть специфику частичных вычислений.

Для этого мы выделим эффекты `Throws` и `Returns` в отдельную группу, и будем называть их **исходами**. Мы будем делить все исходы на **успешные** (`Returns`) и **неуспешные** (`Throws`). Теперь мы можем более аккуратно сформулировать преобразование Seq, которое берет два утверждения, выполняющихся последовательно, и формирует из них одно:

$$Seq(c_1 \to e_1, c_2 \to e_2) = \begin{cases} c_1 \wedge c_2 \to e_1, e_2 & \text{если } e_1 \text{ содержит успешный исход} \\ c_1 \to e_1 & \text{иначе} \end{cases}$$

Таким образом, результирующая схема $\alpha(A, B)$ будем выглядеть следующим образом:

`schema` α`(A, B):`

$$\left[\begin{array}{l} \alpha\Big(Seq(c_1^A \to e_1^A, c_1^B \to e_1^B)\Big) \\ \alpha\Big(Seq(c_1^A \to e_1^A, c_2^B \to e_2^B)\Big) \\ \dots \\ \alpha\Big(Seq(c_1^A \to e_1^A, c_m^B \to e_m^B)\Big) \\ \alpha\Big(Seq(c_2^A \to e_2^A, c_1^B \to e_1^B)\Big) \\ \dots \\ \alpha\Big(Seq(c_n^A \to e_n^A, c_m^B \to e_m^B)\Big) \end{array}\right]$$

2.2.6. Сглаживание утверждений

В целом, процедура сглаживания утверждений очень похожа на процедуру сглаживания операторов, но она обладает рядом чисто технических тонкостей, о которых нельзя не упомянуть.

Пусть дано утверждение, заключением которого является набор эффектов I, а посылкой – схема эффектов A, определенная следующим образом:

`schema A:`

$$\left[\begin{array}{l} c_1^A \to e_1^A \\ c_2^A \to e_2^A \\ \dots \\ c_n^A \to e_n^A \end{array}\right]$$

Мы бы хотели как можно больше переиспользовать алгоритм сглаживания операторов. Заключение I легко выразить в форме эквивалентной схемы эффектов с одним утверждением «`true -> I`», и тогда на это можно смотреть как на еще один бинарный оператор «$\to$». Однако у него есть своя специфика, связанная с тем, что ложная посылка не влечет заключения. Другими словами, при сглаживании утверждений `Returns(false)` является

своего рода *неуспешным* исходом.

При этом нужно учитывать, что утверждения с неуспешным исходом раньше добавлялись в результирующую схему «как есть»: $Seq(c_i^A \to e_i^A, c_j^B \to e_j^B) = Seq(c_i^A \to e_i^A)$, если e_i^A содержит неуспешный исход. Теперь же если этот исход `Returns(false)`, то необходимо аккуратно удалить `Returns(false)` из списка эффектов e_i^A, потому что этот `Returns` не имеет отношения к результирующей схеме.

Если E – список эффектов, а e – отдельный эффект, то будем обозначать $E \setminus e$ список E без эффекта e.

Таким образом, для стрелки импликации Seq_{impl} определен следующим образом

$$Seq_{impl}(c_1 \to e_1, c_2 \to e_2) =$$
$$= \begin{cases} c_1 \wedge c_2 \to e_1, e_2 & \text{если } e_1 \text{ содержит успешный исход} \\ c_1 \to e_1 \setminus Returns(false) & \text{если } e_1 \text{ содержит Returns(false)} \\ c_1 \to e_1 & \text{иначе} \end{cases}$$

2.2.7. Примеры преобразований операторов

Теперь нам осталось лишь определить семантику преобразований операторов. Большинство преобразований устроены очень просто, и причину этого мы уже отметили, когда рассматривали *Checker Framework* – зачастую эффекты приносят с собой некоторые связанные операторы, и вместе образуют замкнутую систему, которая очень мало взаимодействует с остальными.

Таким образом, для значительной части преобразований $\alpha(e_1, e_2, \ldots, e_n)$ большинство эффектов будет прозрачно «подниматься» через преобразование: $\alpha(e_1, e_2, \ldots, e_n) = e_1, e_2, \ldots, \alpha(e_i, e_{i+1}, \ldots, e_j), \ldots e_n$.

Например, операторы `&&`, `||`, `==`, `is`, `!`, `is` образуют естественную группу вместе с эффектами, которые говорят про возвращаемое значение (`Returns`) и работают только с ними. Для них всех преобразование состоит в том, что оператор «проваливается» внутрь эффекта:

$$\texttt{\&\&(Returns(x), Returns(y))} \equiv \texttt{Returns (x \&\& y)}$$
$$\texttt{||(Returns(x), Returns(y))} \equiv \texttt{Returns (x || y)}$$
$$\texttt{==(Returns(x), Returns(y))} \equiv \texttt{Returns (x == y)}$$
$$\texttt{is}_{type}\texttt{(Returns(x))} \equiv \texttt{Returns (x is type)}$$
$$\texttt{!(Returns(x))} \equiv \texttt{Returns (!x)}$$

А вот эффект `Calls` вообще не требует для работы с ним никаких дополнительных операторов – поэтому и правил работы с ним никаких нет, достаточно заложенных в систему алгоритмов комбинации.

Следует понимать, что хотя каждый оператор в отдельности и работает лишь с фиксированной группой эффектов, «интересных» ему, то, что он делает с этой группой и как ее выбирает, остается полностью в его юрисдикции. Выше мы видели, что оператор может выбрать несколько эффектов и слить их в один. Оператор может и удалять эффект (например, если бы ввели оператор `Catch E`, соответствующий аналогичной конструкции – он бы удалял эффект `Throws E` из списка), а может и добавлять. Это следует учитывать при реализации системы эффектов на практике.

2.3. Уменьшение размера схем эффектов

2.3.1. Проблема роста размера схем

Если мы еще раз внимательно взглянем на алгоритм комбинирования схем, представленный в предыдущем разделе, то легко видеть, что при комбинировании схем размера $\mathcal{O}(n)$, $\mathcal{O}(m)$ мы получаем схему размера $\mathcal{O}(n \cdot m)$. Такие комбинирования возникают при вложенных вызовах, но важно понимать, что итоговая длина зависит от устройства схем и *не ограничена* сверху произведением всех использованных схем.

Действительно, рассмотрим функции `f(x, y, z)`, `g()`, `h()`, `r()` (типы опущены для ясности). Пусть размеры схем имеют сложность $\mathcal{O}(F)$, $\mathcal{O}(G)$, $\mathcal{O}(H)$, $\mathcal{O}(R)$ соответственно. Рассмотрим вызов `f(g(), h(), r())`.

Пусть схема для `f` содержит утверждение вида `(x == y) && (y == z)`. После выполнения подстановки это утверждение превращается в

`(g() == h()) && (h() == r())`, которое нуждается в сглаживании.

Сглаживание выражения `g() == h()` дает схему сложностью $\mathcal{O}(G \cdot H)$, а сглаживание выражения `h() == r()` дает схему сложностью $\mathcal{O}(H \cdot R)$. Наконец, сглаживание оператора `==` дает схему сложностью $\mathcal{O}(G \cdot H^2 \cdot R)$. Если каждое утверждение в F имеет подобную структуру, то в итоге получаем схему сложностью $\mathcal{O}(F \cdot G \cdot H^2 \cdot R)$, что *больше*, чем просто произведение всех схем.

Оценку сверху можно получить, если представить схему f в виде дерева с $\mathcal{O}(F)$ узлами, а схему g в виде дерева с $\mathcal{O}(G)$ узлами. Тогда операция подстановки заменяет некоторые узлы в дереве f на дерево g, и верхняя оценка для такой подстановки будет $\mathcal{O}(G^F)$ (заметим, что такая оценка является даже недостижимой, поскольку в реальности в некоторые узлы подстановка не может осуществляться – например, в узлы, соответствующие операторам)

Подобная проблема характерна для условных эффектов при последовательном комбинировании – например, при обзоре литературы мы уже видели экспоненциальный взрыв длины эффектов в статье *Types for Atomicity* [26]. Однако там из-за простого устройства условий (они все имеют вид «Владеет ли функция некоторым набором блокировок?») авторам удалось справиться с этой проблемой. В нашем случае, надеяться на простое решение не стоит, т.к. в представленной системе условия могут представлять собой весьма нетривиальные комбинации из операторов, переменных, вызовов и т.д.

В связи с этим, в этом разделе мы поговорим о двух подходах к уменьшению размера схем. Один из них будет уменьшать размер схем без потери информации и будет, по большому счету, заниматься вычислением констант или приведении эффектов к некоторому каноническому виду. Кроме того, мы опишем второй подход, который потенциально может сжимать схемы намного сильнее, но ценой этому будет потеря информации.

2.3.2. Сокращение схем эффектов

Если в схеме встречаются константы, то можно надеяться, что в схеме можно провести некоторые очевидные сокращения, основанные на знании о том, как операторы работают со своими аргументами.

Например, если у нас есть выражение вида `false && x`, то очевидно, что оно вычисляется в `false` вне зависимости от того, чему равна переменная `x`. Таким образом, мы получаем следующий набор безопасных преобразований:

1. Сокращение логических операторов:

   `false && x` $\equiv$ `false`

   `x && false` $\equiv$ `false`

   `true || x` $\equiv$ `true`

   `x || true` $\equiv$ `true`

   `!true` $\equiv$ `false`

   `!false` $\equiv$ `true`

2. Сокращение оператора проверки на равенство (неравенство):

   Если значения величин x и y точно известны, то `x == y` вычисляется в `true` или `false` в зависимости от того, как сравниваются на равенство соответствующие значения – конкретные правила определяются языком. Значения могут быть известны, если x и y являются константами, или же переменные, про которые можно точно доказать, что они имеют то или иное значение.

3. Сокращение оператора проверки на подтипизацию:

   `x is T` может быть сокращено в `true` или `false`, если можно доказать, что `x` точно является или не является подтипом типа `T`. Также как и с оператором равенства, мы всегда знаем точный тип констант, но вот с переменными нужно быть аккуратнее. Конкретные правила зависят от конкретной системы типов – например, в языках с полиморфизмом нужно помнить, что статический тип переменной может отличаться от динамического.

4. Сокращение стрелки импликации в утверждениях:

   Как мы уже говорили, ложные посылки не влекут следствие, поэтому утверждения вида «`false` $\to$ `E`» можно удалить из схемы эффектов

Может показаться, что если в вызове не фигурируют константы, то пользы от этого подхода нет совсем, но на самом деле это не так – константы могут появляться из схемы эффектов. Например, схема для функции `assert(x)` содержит выражения вида `x == true`. Если вместо `x` подставляется некоторая другая схема эффектов, в которой есть утверждения вида `Returns(false)` или `Returns(true)`, то в схеме, полученной после сглаживания, мы вполне можем увидить выражения вида `false == true` или `true == true`.

Таким образом, данный подход является вполне работоспособным для схем эффектов, которые используют константы – и практика показывает, что такие схемы встречаются весьма часто.

Кроме того, заметим, что используя более сложные алгоритмы, можно развивать этот подход. Дело в том, что некоторые утверждение не являются независимыми – например, выражения `x is String` и `x !is String` не могут быть верными одновременно, поэтому выражение вида `(x is String) && (x !is String)` можно безопасно сократить до константы `false`. В данной работе мы не будем рассматривать подобные алгоритмы, отсылая читателя к классическим источникам [16] или же более современным работам [13]

2.3.3. Аппроксимация схем эффектов

Однако иногда сокращения схем оказывается недостаточно. Рассмотрим достаточно логичный и жизненный пример – пусть есть какой-то очень сложный вызов с большой вложенностью, в недрах которого используется простая функция `assertNonNull(x)`, которая содержит утверждение
`x == null` $\rightarrow$ `Throws AssertionError`. Будем считать, что никакая другая функция не обрабатывает это исключение – тогда и для всего вызова данное утверждение является корректным. Тем не менее, в силу сглаживания, это утверждение будет появляться повсюду вместе с некоторыми другими условиями и эффектами:

schema :

$$\left[\begin{array}{l} \ldots \\ \texttt{x == null \&\&}\ c_1 \rightarrow \texttt{Throws AssertionError,}\ e_1 \\ \texttt{x == null \&\&}\ \overline{c_1} \rightarrow \texttt{Throws AssertionError,}\ e_2 \\ \ldots \end{array}\right]$$

Для данного вызова верно, что если `x == null`, то всегда генерируется исключение `AssertionError`, но эта информация выражена в схеме неявно из-за дополнительных условий и эффектов. На практике, простые и краткие утверждения являются намного более полезными и удобными, чем длинные и подробные. Этому есть некоторое интуитивное обоснование.

Пусть весь вызов задействует переменные $x_1, x_2, \ldots x_n$, каждая из которых может принимать $D_1, D_2, \ldots D_n$ различных значений. Рассмотрим некоторое утверждение. Чем больше переменных задействовано в его посылке, тем меньшую долю всех входов покрывает это утверждение. Получается, что сами по себе длинные утверждения требуют больше памяти для хранения, больше времени для работы с ними, в то время как реально использоваться они будут реже.

Для примера рассмотрим частный случай, когда x_1 является булевской переменной и $D_1 = 2$. Тогда утверждение, посылка которого имеет вид $\texttt{x}_1$ `== true`, выполняется для половины всех возможных наборов входных параметров, т.е. грубо говоря, оно будет полезно в каждом втором случае.

Это неформальное рассуждение должно показать, что на практике вполне полезным преобразованием является замена множества длинных утверждений на одно короткое, пусть даже при этом мы потеряем некоторую информацию. Это преобразование мы будем называть **аппроксимацией** схем эффектов.

В целом, эта задача имеет непосредственное отношение к задаче аппроксимации булевых функций, для которой разработаны достаточно эффективные решения [3]. Здесь мы рассмотрим наиболее простой в реализации, не вдаваясь в подробности его свойств и сравнения с другими, существующими алгоритмами. Однако даже не смотря на его простоту, он требует введения ряда понятий и терминов, в которых можно легко «потеряться». Поэтому мы будем рассматривать его работу на примере следующей схемы:

```
schema example:
⎡ x is String && x !is String → Returns(true)                  ⎤
⎢ y == null && b == true → Throws IllegalArgumentException     ⎥
⎢ y == null && b == false → Throws IllegalArgumentException    ⎥
⎣ y != null && b == true → Returns(false)                      ⎦
```

Мы будем время от времени возвращаться к этой схеме, чтобы продемонстрировать некоторые понятия на ее примере. Подобные отступления будут выделены текстом в рамке.

Идея алгоритма состоит в следующем. Рассмотрим схему после сглаживания – тогда посылка любого утверждения представляет собой набор операторов, соединенных логическими связками (отрицание, конъюнкция, дизъюнкция). Операторы, не являющиеся логическими связками, мы заменим на логические переменные $x_1, x_2, \ldots, x_n$. Одинаковым по содержанию операторам будут соответствовать одинаковые переменные.

Кроме того, совместность логических переменных должна по возможности соответствовать совместности логических операторов – т.е. если выражению `x is String` была назначена переменная x_1, то выражению `x !is String` должна быть назначена переменная $\overline{x_1}$, а не x_2. Разумеется, некоторые связи могут быть достаточно сложными для того, чтобы их так просто выявить, но с практической точки зрения, вполне достаточно рассмотреть простые случаи типа `is/!is`, `==/!=` и `true/false`.

Аналогично мы поступим и со всеми эффектами. После этого мы можем привести посылку к дизъюнктивной нормальной форме, и утверждения в схеме будут иметь вид логических формул вида $DNF(x_1, \ldots, x_n) \to e$.

Для схемы *example* мы заменяем операторы и эффекты на переменные следующим образом:

`x is String`	x_1
`x !is String`	$\overline{x_1}$
`y == null`	x_2
`b == true`	x_3
`b == false`	$\overline{x_3}$
`y != null`	$\overline{x_2}$
`Returns(true)`	e_1
`Throws IllegalArgumentException`	e_2
`Returns(false)`	e_3

После этого получаем следующий набор логических формул:

$$x_1 \wedge \overline{x_1} \to e_1$$
$$x_2 \wedge x_3 \to e_2$$
$$x_2 \wedge \overline{x_5} \to e_2$$
$$\overline{x_2} \wedge x_3 \to e_3$$

Мы фиксируем набор S интересующих нас булевых функций.

Для нашего примера мы рассмотрим две функции: одноместную функцию Id, соответствующую тождественной функции, и нульместную функцию $False$, соответствующую тождественно ложной функции.

После этого мы перебираем все возможные наборы значений для всех входных переменных, и для каждого набора выписываем, какие эффекты были сгенерированы при таких значениях, а какие нет. Интуитивно, мы получаем нечто вроде таблицы истинности для нескольких функций, каждая из которых соответствует какому-то эффекту. Наша дальнейшая задача будет заключаться в том, чтобы оставить только те эффекты, которые могут быть выражены через достаточно простые функции – т.е., через функции из набора S.

x_1	x_2	x_3	e_1	e_2	e_3
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	1	0
0	1	1	0	1	0
1	0	0	0	0	0
1	0	1	0	0	0
1	1	0	0	1	0
1	1	1	0	1	0

Для того, чтобы найти простые зависимости, мы перебираем все функции из S. Пусть мы зафиксировали функцию s арности r. Мы бы сейчас хотели узнать – нет ли в нашей схеме каких-нибудь r переменных и какого-то эффекта, таких, что их связь описывается в точности функцией s.

Давайте перебирать все наборы из r переменных и эффекта. Зафкисируем пару $(x_{i_1}, \ldots, x_{i_r}; e_j)$. Как проверить, что зависимость между этими переменными и эффектом в действительности описывается функцией s? Интуитивно кажется, что для этого нужно проверить, что наличие либо отсутствие эффекта в схеме согласовано с выводом функции s для любого набора значений переменных.

Формулизуем это. Переберем 2^r возможных значений набора переменных $x_{i_1}, \ldots, x_{i_r}$. Для каждого набора есть его **результат функции s на этом наборе**, который определяется тривиальным вычислением s на этих аргументах.

Результат же этого набора в схеме относительно эффекта e_j определяется чуть-чуть более сложнее. Возьмем набор значений, и взглянем на те строчки, где он появляется. Если *во всех* этих строчках эффект e_j имеет одно и то же значение, то результатом этого набора относительно эффекта e_j является это значение. В противном случае, этот набор значений *не определяет* эффект e_j, и мы будем говорить, что результатом этого набора является **неизвестность**

Например, результатом набора значений $x_2 = true, x_3 = false$ относительно эффекта e_2 является истина, поскольку в обоих строчках, в которых появляется этот набор, данный эффект имеет значение 1.

А вот набор значений $x_3 = true$ относительно этого же самого эффекта имеет значение «неизвестность», поскольку, например, при $x_1 = 0, x_2 = 1, x_3 = 1$ получаем $e_2 = 1$, но при $x_1 = 0, x_2 = 0, x_3 = 1$ получаем $e_2 = 0$.

Мы будем говорить, что пара $(x_{i_1}, \ldots, x_{i_r}; e_j)$ **описывается** функцией s, если для любого набора значений переменных $x_{i_1}, \ldots, x_{i_r}$ результат функции на этом наборе *в точности* совпадает с результатом в схеме относительно эффекта e_j.

Нетрудно видеть, что если пара $(x_{i_1}, \ldots, x_{i_r}; e_j)$ описывается функцией s, то мы можем добавить в результирующую схему утверждение вида $s(x_{i_1}, \ldots, x_{i_r}) \to e_j$.

Например, в рассматриваемом примере эффект пара $(\{x_2\}; e_2)$ описывается функцией Id. Пара $(\emptyset; e_1)$ описывается функцией $False$, что означает, что эффект e_1 никогда не генерируется.

Мы не будем выписывать псевдокод данного алгоритма, поскольку выше мы привели очень подробное описание с примерами. Отметим, что построение таблицы истинности экспоненциально по количеству переменных в схеме. В свою очередь, потенциально это количество пропорционально размеру схемы. Может показаться, что данный алгоритм не особо решил проблему, поскольку экспонента в одном месте (при комбинации схем) была заменена экспоненту в другом месте (при аппроксимации). Тем не менее, на практике выигрыш получается довольно ощутимым за счет того, что после нескольких сглаживаний мы можем легко получить очень большую (по сравнению с исходной) схему, в то время как переменных в ней будет использоваться по-прежнему не так много.

3. Практические аспекты использования систем условных эффектов

3.1. Извлечение информации из схем эффектов

3.1.1. Два направления вывода информации

По итогам прошлой главы, мы научились получать схему эффектов для произвольного вызова (при условии, конечно, что все участвующие в нем функции аннотированы схемами). Теперь мы хотели бы научиться извлекать из полученных схем полезную информацию.

В целом, мы можем попытаться извлечь информацию из схемы с помощью двух различных подходов:

- Зная некоторую информацию о контексте исполнения, попытаться понять, какие посылки исполняются и какие эффекты может иметь вычисление. Мы будем называть этот подход **прямым направлением вывода**, т.к. он сонаправлен с потоком вычислений в программе.

- Зная, что подпрограмма сгенерировала некоторые эффекты, попытаться понять, какой вид имел контекст исполнения. Мы будем называть этот подход **обратным направлением вывода**, т.к. он противонаправлен потоку вычислений в программе.

Рассмотрим для примера следующую схему:

```
schema assertIsNull(x):
⎡ x == null → Returns Unit           ⎤
⎣ x != null → Throws AssertionError  ⎦
```

Тогда мы можем извлечь некоторую пользу из этой схемы двумя способами, соответствующими описанным выше подходами:

- Понять, что пользователь передает в эту функцию переменную, которая гарантированно не является `null`, и предупредить о том, что это действие наверняка вызовет исключение. Как следствие, весь последующий код в данном блоке является недостижимым, о чем также можно сообщить, если это поддерживается инструментами разработки:

```
1  ...
2  assertIsNull("Constant string can not be null")
3  println("This statement is unreachable!")
4  ...
```

- Мы можем зайти и с другого конца. Можно посмотреть на следующее после вызова выражение и понять, что вычисление может дойти до него только если исключение не было сгенерировано, т.е. если аргумент, переданный в `assertIsNull`, не является `null`

```
1  val s: String?
2  ... <initialize s> ...
3
4  assertIsNull(s)
5  println(s.length)  // here we can be sure that 's != null'
```

Также как и в предыдущем примере, для того, чтобы конечный пользователь ощутил реальную пользу от этого анализа, необходимо, чтобы инструмент поддерживал некоторый механизм, для которого такая информация была бы полезна. Для этого подхода отлично подходит механизм умных приведений типа в `Kotlin` – с его помощью мы можем автоматически уточнить тип переменной `s` из последнего примера.

Алгоритм для прямого вывода достаточно прост и интуитивен – нужно просто вычислить все операторы в данном контексте, и затем аккуратно собрать эффекты. Некоторые нетривиальности могут возникнуть при обработке посылок, которые оказались вычисленными не до конца, т.е. когда имеющейся в распоряжении информации о контексте не хватает для того, чтобы сделать вывод о том, верна эта посылка или же нет. Однако в таком случае сложно предложить какой-то общий подход к разрешению подобных ситуаций, поскольку это тесно связано с понятием консервативного приближения, и, как следствие, конкретного типа анализа.

В связи с этим, мы не будем заострять внимание на прямом выводе, а вместо этого подробнее рассмотрим алгоритм обратного вывода ниже.

3.1.2. Алгоритм обратного вывода

Алгоритм обратного вывода состоит из нескольких подчастей:

1. Операция **фильтрации**, который оставляет в схеме только интересные нам выражения. Например, если мы хотим узнать вид контекста при условии, что функция успешно завершилась, то этому алгоритму будет передано выражение `Returns ???`, и он вернет только те утверждения, которые имеют в заключении эффект `Returns`.

2. Операция **слияния**, который принимает на вход набор утверждений, полученных после фильтрации, и объединяет информацию, заложенную в этих утверждениях.

Начнем с алгоритма фильтрации. Для того, чтобы понять, как он устроен, опишем более формально то, что он должен делать:

Определение 3.1. Пусть дано выражение q и утверждение s. Будем говорить, что q **влечет** s (s **следует из** q), если информации о том, что были сгенерированы эффекты из заключения s достаточно для того, чтобы сделать вывод, что имел место эффект q. Далее мы будем говорить, что $s_{effects}$ влечет q (q следует из $s_{effects}$)

Поясним это примером. Рассмотрим схему:

```
schema A:
[ x is String → Returns(true)  ]
[ y != null → Calls(f, 1)      ]
```

Рассмотрим первое утверждение из схемы. Оно, очевидно, следует из выражения `Returns true`. Но что более интересно, оно следует также и из выражения `Returns ???` – действительно, если заключение утверждения верно (т.е. функция вернула `true`), то, конечно, верно, что функция завершилась (т.е. буквально `Returns ???`)

Теперь, вооружившись отношением «влечет» и дуальным к нему «следует», мы можем описать суть алгоритма фильтрации:

Определение 3.2. Операция фильтрации схемы S по выражению q выдает схему S' состоящую только из таких утверждений S, которые следуют из q. Будем обозначать эту операцию как $Filter(S, q)$

Перейдем теперь к операции слияния. Будем считать, что нам дана некоторая схема S вычисления f, и необходимо собрать информацию, заложенную в ее утверждениях. Введем понятие **приближенного контекста** для f – это набор информации об окружении f (типов и значений переменных, счетчиков вызовов функций и т.д.), являющейся корректным консервативным приближением реального контекста f. Будем обозначать приближенный контекст для f как $\mathcal{C}(f)$.

Очень важно отметить, что для разных видов анализа вид и форма информации, хранящейся в $\mathcal{C}(f)$, может быть разной.

Мы бы хотели научиться строить максимальный приближенный контекст для схемы S. Разумеется, делать мы это будем, поднимаясь снизу вверх по дереву иерархии:

```
1  Input:  root - вершина дерева схемы эффектов
2  Output: максимальный приближенный контекст для поддерева с корнем root
3
4  Algorithm GetContextApproximation(root) {
5    n ← root.childs.size
6    for (i in 1..n) {
7      v ← root.childs[i]
8      C_i ← GetContextApproximation(v)
9    }
10
11   return JoinContexts(root, C_1, C_2, ..., C_n)
12 }
```

Легко видеть структурную схожесть с алгоритмом сглаживания – мы также используем многошаговую рекурсивную процедуру, которая внутри,

после рекурсивных вызовов, использует одношаговую процедуру для объединения результатов.

Таким образом, необходимо понять, как устроено дерево, на котором будет вызываться $GetContextApproximation$, определить операцию $JoinContext$ на его листьях и всех внутренних узлах.

По определению, $GetContextApproximation$ вызывается только на плоских схемах. Как мы уже видели, посылка любого утверждения в такой схеме является набором операторов `is`, `==`, `!is`, `!=`, связанных булевыми операторами. Таким образом, на листьях $JoinContext$ определяется очевидно, выдавая просто контекст, хранящий соответствующую информацию о типе или значении переменной. Напомним, что конкретная форма этой информации определяется анализом.

Намного интересней это преобразование устроено на бинарных логических операторах, а именно на логическим «ИЛИ» и логическом «И». Комбинируя два контекста, связанных оператором «ИЛИ», необходимо учитывать, что какой-то из этих двух контекстов может оказаться неверным. Аналогично, при комбинации двух контекстов, связанных оператором «И», нужно понимать, что оба контекста являются верными.

К сожалению, в отрыве от конкретного типа анализа, невозможно определить эти преобразования явно. Лучшее, что мы можем здесь сделать, это потребовать, чтобы каждый вид приближенных конекстов определял операции `or` и `and`, соответствующие комбинации двух контекстов с семантикой «ИЛИ» и «И», описанной выше.

Буквально в следующем разделе мы будем говорить о конкретных применениях систем эффектов, и там мы сможем конкретно сформулировать эти преобразования. Пока что же отметим еще несколько нюансов:

- $JoinContexts\big(statement, \mathcal{C}(premise), \mathcal{C}(conclusion)\big)$, где $statement$ – это утверждение, а $premise$ и $conclusion$ – его посылка и заключение соответственно, выдает просто $\mathcal{C}(premise)$ and $\mathcal{C}(conclusion)$. Действительно, все утверждения, собранные в посылке и заключении, верны *одновременно*

- $JoinContexts\big(schema, \mathcal{C}(s_1), \mathcal{C}(s_2), \ldots, \mathcal{C}(s_n)\big)$, где $schema$ – это схема эф-

фектов, а $\mathcal{C}(s_i)$ – ее утверждения, выдает контекст, соответствующий $\mathcal{C}(s_1)$ *or* $\mathcal{C}(s_2)$ *or* ... *or* $\mathcal{C}(s_n)$. Действительно, утверждения схемы являются независимыми, поэтому мы не можем утверждать, что все они выполняются одновременно.

Важно отметить, что обычно мы не можем гарантировать и то, что хотя бы одно из них выполняется, в то время как семантика преобразования *or* требует выполнения такого предусловия. Однако именно поэтому нам нужен был шаг фильтрации, который оставляет такой набор утверждений, что по крайней мере одно из них выполняется.

- Практика показывает, что удобней «спустить» все отрицания до примитивных операторов типа `!is` и `!=` с помощью закона де Моргана, нежели определять семантику отрицания контекстов – для некоторых видов контекста это может быть довольно нетривиальной операцией. Мы будем пользоваться этим соглашением в будущем.

3.2. Полный алгоритм работы системы эффектов

В данном разделе мы представим пример использования системы эффектов, собирая воедино все вышесказанные алгоритмы, а также описывая некоторые технические детали, связанные с интеграцией системы эффектов в язык.

Работа с системой эффектов осуществляется в несколько этапов. На каждом этапе имеется некоторая отдельная структура данных, и между двумя последовательными этапами существует преобразование, переводящее одну структуру в другую. Ниже мы вкртаце опишем каждую структуру и преобразование, использованные в данной работе (и, соответственно, использованные при реализации системы в языке `Kotlin`)

1. **AST-дерево**. Оно отражает структуру кода на уровне синтаксиса – например, хранит в себе различные литералы, вроде скобок и запятые. Дополнительная информация (например, соответствующее вызову определение функции) хранится отдельно, либо может даже вовсе отсутствовать.

2. **Дерево вызовов** – отражает структуру кода на более удобном для использовании в схеме эффектов уровне абстракции. Это довольно простая древовидная структура, каждый внутренний узел в которой – некоторый вызов, а его дети – аргументы этого вызова. Листьями дерева являются константы и переменные. В дереве вызовов хранится вся интересная для системы эффектов информация (в частности, соответствующие функциям схемы), и только она.

   Помимо чисто технических преобразований, на этом этапе необходимо откуда-то получить схемы эффектов. В данной работе схемы эффектов или выводятся автоматически (для простых выражений), или же парсятся из строковых аннотаций с помощью ANTLR.

3. **Дерево схем** – по сути, это схема, в которой могут существовать вложенные схемы. Данная структура получается из дерева вызовов заменой каждого узла-вызова на соответствующую схему эффектов, в которой выполнена операция подстановки, о которой мы говорили в пункте 2.2.1.

4. **Плоская схема эффектов** – схема, в которой не может быть вложенных схем эффектов. Преобразование из дерева схем в точности соответствует оператору сглаживания, подробно обсуждавшемуся в главе 2.

5. **Сокращенная схема эффектов** – при необходимости, для того, чтобы не допускать чрезмерного роста размера схем, после предыдущего шага выполняются преобразования сокращения и аппроксимации схем, описанные в разделе 2.3,

6. **Набор информации о контексте** – набор информации, которая описывает эффекты выражения при условии, что известна некоторая информация о контексте или наблюдаемых эффектах. Важно отметить, что этот набор информации должен быть пригоден для использования в компиляторе, т.е. эта структура является совместным интерфейсом компилятора и системы эффектов.

Переход от плоской схемы эффектов к набору информации о контексте подробно описан в пункте 3.1.

Рассмотрим пример: пусть имеется следующий вызов

```
1  foo(1, bar(s, null))
```

Пусть мы хотим получить информацию о подтипах переменных в том случае, если этот вызов успешно завершился. Тогда на рисунках 1, 2, 3, 4, 5, 6 показаны описанные выше стадии для данного примера:

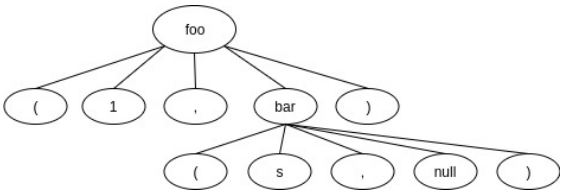

Рис. 1: AST-дерево

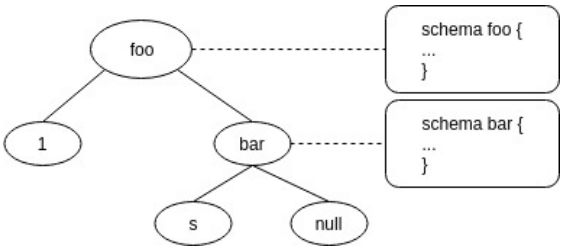

Рис. 2: Дерево вызовов

```
schema foo:
[ 1 == null → Throws IllegalArgumentException
  schema bar:
  [ s is String → Returns (s + "1")
    s !is String → Returns (null) ]  == null → Returns (Unit) ]
```

Рис. 3: Дерево схем

```
schema foo:
[ 1 == null → Throws IllegalArgumentException
| s is String && (s + "!") == null → Returns (Unit)
[ s !is String && null == null → Returns (Unit)
```

Рис. 4: Плоская схема

```
schema foo:
[ s !is String → Returns (Unit) ]
```

Рис. 5: Сокращенная схема

```
s !is String
```

Рис. 6: Результирующая информация

3.3. Применение системы эффектов в компиляторе Kotlin

3.3.1. Основы анализа потока данных

Статический анализ в `Kotlin` тесно связан с фреймворком анализа потока данных (англ. *data-flow analysis*). Это довольно обширная и глубокая тема, и пересказать ее в рамках работы не представляется возможным. Тем не менее, нам необходимо обрисовать эту концепцию хотя бы в общих чертах, поскольку в противном случае разговор об улучшении анализа в `Kotlin` будет слишком неконкретным.

Начнем с понятия графа потока управления (англ. *control-flow graph, CFG*). Каждая инструкция в нем представлена одной вершиной, и если между вершинами u и v есть ребро, то это означает что после инструкции u управление может быть передано в инструкцию v.

Рассмотрим простой пример кода:

```
1  if (x == 0) {
2      println("True branch")
```

```
3  } else {
4      println("False branch")
5  }
6  println("If-end")
```

Ему соответствует граф потока управления, как на рисунке 7

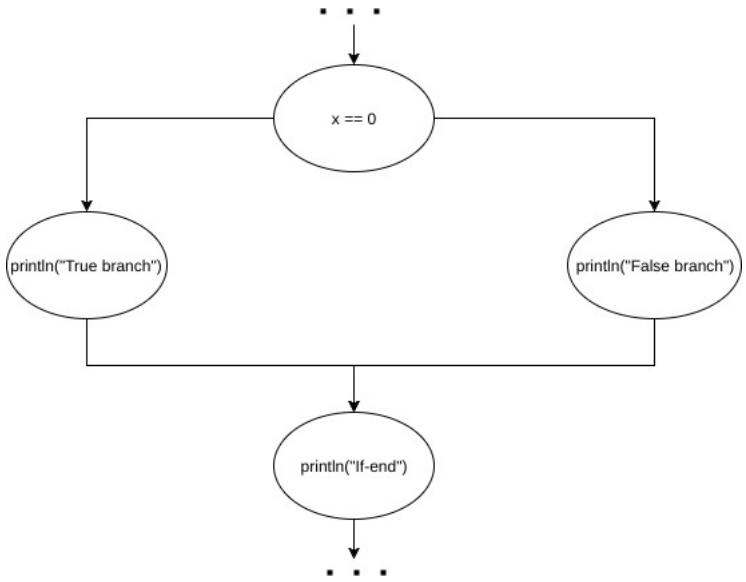

Рис. 7: Пример графа потока управления

Так, после инструкции, вычисляющей условие в `if`, поток управления раздвоился, отражая тот факт, что мы не можем знать при статическом анализе (без дополнительных предположений), какая из веток выполнится. После ветки вновь сливаются в один поток, как и следовало ожидать.

Разумеется, граф потока управления не обязан быть ациклличным – конструкции `for`, `while`, `goto` (безусловный переход) могут вносить в него обратные ребра.

Важно заметить, что любому возможному пути исполнения в программе обязательно соответствует некоторый путь в CFG, но обратное не верно. Например, путь может проходить через истинную ветку выражения `if (x == 0)` и через истинную ветку выражения `if (x != 0)`, хотя между первым и вторым оператором могло и не быть никаких изменений переменной `x`, и иногда это даже можно доказать статически. Тем не менее, это не

нарушает консервативности анализа, поскольку при анализе будет обязательно рассмотрен любой реально возможный путь исполнения.

Конструкция графов потока управления сама по себе позволяет обнаружить совсем простые ошибки, вроде недостижимого кода из-за неправильного использования безусловных переходов. Однако она становится особо мощной и полезной, если использовать ее вместе с концепцией анализа потока данных.

Определение 3.3. Анализ потока данных (англ. *data-flow analysis*) – это метод статического анализа, который основывается на извлечении информации из характеристик и свойств потока данных вдоль различных путей исполнения в программе.

Основная идея основывается на наблюдении, что в любой момент времени при выполнении программы существует некоторое глобальное состояние, которое состоит из множества всех переменных, их значений, а также другой информации, зависящей от конкретного типа анализа (например, счетчик количества вызовов для функций, статус инициализации переменной, и т.д.). Тогда для каждой точки программы можно ввести понятие **значения потока данных** (от англ. *data-flow value*), которое является абстракцией всех возможных глобальных состояний, которые можно наблюдать в данной точке. Для краткости, в дальнейшем мы будем писать DFV вместо «значение потока данных».

В силу того, что потенциально количество возможных путей исполнения в программе может быть бесконечно [1], на практике делается два упрощения: во-первых, конкретное DFV не хранит историю, как управление могло придти к этой точке программы, а во-вторых, в зависимости от конкретного анализа, откидывается некоторая излишняя информация. Так, например, при анализе инициализации переменных, нам не важно, какие значения может иметь переменная, и на каких путях исполнения они могли быть получены. Достаточно знать, правда ли, что на любом пути исполнения, достигающем данную точку, данная переменная была инициализирована, или нет. Таким образом, для каждой переменной достаточно просто хранить бинарный флаг, что значительно упрощает реализацию на практике.

Мы не будем вдаваться в подробности того, как вычислять значения DFV, т.к. это выходит за рамки данной работы. Подробней про методы решения систем уравнений на поток данных можно прочитать в канонических источниках: [1, 18]. В дальнейшем нам будет достаточно концепции графа потока управления и значения потока данных (DFV).

3.3.2. Автоматическое приведение типов

Как мы уже говорили, `Kotlin` поддерживает автоматические приведения переменной к более частному типу там, где это возможно. Чаще всего это возможно потому, что если управление дошло до определенной точки в программе, то должны выполняться некоторые ограничения на контекст.

Анализ умных приведений типов полностью основывается на анализе потока данных. Проблемы, описанные в главе 1, связаны с тем, что при извлечении из каждой инструкции начального DFV не учитываются межпроцедурные взаимодействия. Как раз здесь и может помочь система эффектов.

В качестве DFV для анализа умных приведений типов используется специальный класс `DataFlowInfo`, который хранит типовую информацию о переменных из контекста (обратите внимание на небольшую рассинхронизацию терминологии). Кроме того, данный класс поддерживает операции `or` и `and`, в точности соответствующие операциям, определенным нами в прошлом разделе для приближенных контекстов. Таким образом, этот класс естественно подходит на роль приближенного контекста.

Наконец, данный класс уже реализует всю необходимую логику для добавления и корректной обработки информации о подтипизации.

В итоге, в данном пункте с точки зрения системы эффектов не требуется никакой дополнительной работы – необходимо взять выражение и выполнить серию преобразований, описанных в предыдущем подразделе, используя `DataFlowInfo` в качестве приближенных контекстов. Фреймворк анализа потока данных сам обеспечит, чтобы полученная из системы эффектов информация была применена в нужном месте.

3.3.3. Анализ инициализации переменных

Анализ инициализации переменных также построен на основе анализа потока данных. Однако здесь для передачи информации из системы эффектов в компилятор нам понадобятся некоторые дополнительные усилия.

Для начала, вспомним, что проблемы с инициализацией появились из-за того, что у компилятора отсутствовала информация о том, что некоторые функции вызывают другие детерминированное число раз. Однако с точки зрения инициализации переменных, нас интересуют не точные счетчики вызовов каких-то процедур, а более общие понятия, которые мы будем называть **статусом вызовов**:

- `UNKNOWN`, т.е. количество вызовов неизвестно
- `NOT INVOKED`, т.е. ровно ноль вызовов
- `EXACTLY ONCE`, т.е. ровно один вызов
- `AT LEAST ONCE`, т.е. по меньшей мере один вызов
- `AT MOST ONCE`, т.е. не более одного вызова

Таким образом, приближенный контекст для анализа инициализации переменных выглядит как отображение из функций в соответствующие им статусы вызовов. Преобразования `or` и `and` на статусах вызовов определяются довольно безыдейным перебором случаев. Их количество которых может быть несколько оптимизировано при более аккуратном подходе, но это в любом случае остается довольно технической работой, поэтому эти правила вынесены в приложение <TODO: INSERT APPENDIX REF HERE>.

Рассмотрим в качестве примера уже известный нам отрывок кода:

```
1  val x: Int
2  run {
3    x = 42
4  }
5  println(x)
```

Напомним, что изначально такой код не компилировался, поскольку компилятор считал переменную `x` в выражении `println(x)` не инициализированной. Это происходило из-за того, что для функции `run` не было гарантировано, что переданная в нее лямбда `{ x = 42}` будет вызвана ровно один раз.

С точки зрения системы эффектов, для вызова функции `run` мы можем узнать, что переданная в нее лямбда была вызвана ровно один раз благодаря полученному статусу вызовов `EXACTLY ONCE`. Однако передать эту информацию в компилятор не так и просто.

Дело в том, что информация, необходимая для инициализации переменных, выражена в данном случае неявно (сравните это с автоматическим приведением типов, где информация о подтипизации была выражена явно). Если бы мы каким-то образом могли извлекать более явный эффект, например, `writes` (означающий, что функция пишет в некоторую переменную), то процесс интеграции был бы ничуть не сложней, чем в случае с приведениями типов.

Теоретически, можно было бы попытаться вывести этот эффект из тела лямбды, а затем аккуратно скомбинировать с эффектом `Calls`. Однако с точки зрения дизайна это является некоторым мошенничеством – инициализация переменной на самом деле происходит в строке `x = 42`, в то время как мы приписываем инициализацию в строку вызова `run`. Из-за этого могут возникать чисто технические неприятности – например, если `x` и так уже была инициализирована, то ошибка повторной инициализации будет выдана с неправильным номером строки.

Поэтому мы постараемся поступить более «честно» – а именно, донести до компилятора информацию, что поток управления заходит в лямбду и затем выходит из нее, возвращаясь в точку вызова. Иными словами, мы бы хотели добавить ребро из инструкции вызова в начало декларации лямбды, а из конца декларации – в следующую после вызова инструкцию (см. рисунок 8)

Этот подход, однако, значительно усложняется, если вместо лямбды используется именованная функция. Рассмотрим, например, следующий пример:

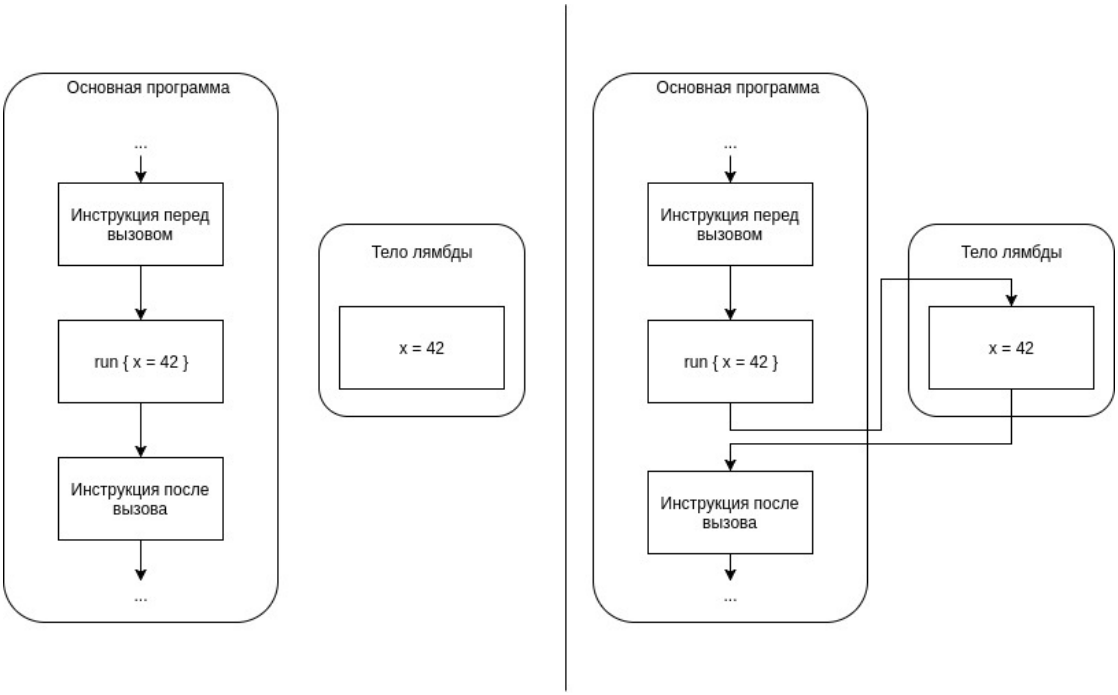

Рис. 8: Преобразование потока управления.
Слева – исходная ситуация, справа – после внесения изменений

```
1  val x: Int
2  val block = { x = 42 }
3
4  println(x)  // Not initialized yet
5  run(block)
6  println(x)  // Initialized once
7  run(block)
8  println(x)  // Re-initialization
```

Именованную функцию можно вызывать несколько раз из различных участков кода – и каждый вызов может иметь различное значение с точки зрения анализа инициализации. Например, в приведенном выше примере первый вызов корректно инициализирует переменную `x`, а вот второй вызов уже вызывает ошибку из-за повторной инициализации. Но поток управления в обоих случаях проходит по одной и той же инструкции, что может опять

привести к проблемам с диагностическими сообщениями, и т.д.

В связи с этим, в рассматриваемой реализации было принято решение ограничиться рассмотрением только анонимных функций. Подчеркнем, что это не является недостатком разработанной системы, а, скорее, некоторыми техническими ограничениями предметной области, не позволяющими в полной мере воспользоваться информацией, поставляемой системой эффектов.

Кроме того, необходимо доставлять в компилятор информацию об `AT LEAST ONCE`, `NOT INVOKED` и `AT MOST ONCE`-вызовах. Первый наиболее актуален, т.к. подобные вызовы могут быть использованы для корректной инициализации `var`-переменных, допускающих повторное присваивание. Подробное описание того, как это можно сделать, в большей мере относится к анализу потока данных, нежели к системам эффектов, и потому выходит за рамки данной работы. Отметим, однако, что с точки зрения анализа потока данных, `AT LEAST ONCE` соответствует циклу `do-while` с условием, про которое не известно, сколько раз оно выполнится – на этом наблюдении и основываются изменения, вносимые в поток управления.

3.4. Реализация приведений типов в коллекциях

Напомним, что изначально мы упоминали еще одну проблему, которую мы пока что никак не решили – это автоматическое приведение типов в коллекциях, например в вызовах типа `list.filter{ x -> x is String }` .

Мы не случайно отложили этот вопрос напоследок. Мы уже упоминали, что системы эффектов наиболее хорошо работают со свойствами функций, которые могут быть кратко выражены. Давайте аккуратно сформулируем, каково свойство вызова `filter`:

«Вызов `filter` возвращает список, элементами которого являются те элементы исходного списка, на которых переданный предикат вернул `true` »

Это довольно длинная мысль, и она никак не похожа на «свойство функции, которое может быть выражено кратко». Если постараться записать это в более формальном синтаксисе, то можно неожиданно придти к теории множеств:

$$S.filter(p) = \{x \in S | p(x)\}$$

Эта запись говорит: `S.filter(p)` выдает множество x из S таких, что верен $p(x)$.

Таким образом, для задачи приведения типов в коллекциях лучше всего подходит формализм теории множеств, а не эффектов. Тем не менее, в данной работе мы покажем, как все-таки реализовать поддержку таких понятий в рамках системы эффектов. Помимо чисто научного интереса, это позволит лучше понять границы применимости системы эффектов и того, насколько сложно добавить в описанный фреймворк не очень свойственные ему понятия.

В дальнейшем мы будем использовать для примера функцию `filter`. Для упрощения мы будем рассматривать следующую ее сигнатуру:

```
1 fun <T> filter(l: Collection<T>, p: (T) -> Boolean): List<T> {
2   ...
3 }
```

Уточнение возвращаемого типа Начнем с того, что пока что в нашей системе нет вообще никакого способа сказать, что функция возвращает более конкретный тип, чем описанный в ее сигнатуре. Легче всего это сделать, если расширить семантику `Returns`, позволив ему возвращать не только значения, но и типы. Запись `Returns(T)` следует понимать следующим образом: функция возвращает *неизвестное* значение типа T.

Заметим, что в предложенной в данной работе реализации по некоторым чисто техническим причинам был выбран другой путь, и был введен эффект `Hints(variable, type)`, который является полным налогом выражения `variable is type`. Их различие состоит в том, что `is` может появляться только слева от стрелки импликации, а `Hints` – только справа. В дальнейшем мы будем пользоваться синтаксисом с перегруженным `Returns`, дабы не плодить излишние сущности.

Таким образом, мы уже можем написать часть схемы для `filter`, которая выглядит следующим образом:

```
schema filter(l, p):
[ true → Returns ( List < ... > ) ]
```

Теперь наша задача состоит в том, чтобы вместо многоточия вставить тип, определяемый поведением лямбды, т.е. тип аргумента лямбды, при условии, что предикат вернул `true`.

Оператор at. Пусть у нас имеется схема эффектов для предиката `p`. Нам необходимо как-то выразить ту мысль, что нас будет интересовать только та часть схемы, которая соответствует случаю, когда предикат вернул `true`. Если внимательно посмотреть на эту формулировку, то можно заметить, что она *в точности повторяет операцию фильтрации схемы*, которую мы определили в пункте 3.1.2.

Таким образом, нам нужно просто предоставить пользователем возможность пользоваться операцией, которая до этого была внутренней для системы эффектов. Для этого мы вводим бинарный оператор `at`. Выражение `S at E`, где `S` – схема, а `E` – эффект, возвращает схему, соответствующую схеме `S` отфильтрованной по эффекту `E` как было определено в пункте 3.1.2.

Теперь мы можем еще на шаг продвинуться к спецификации `filter`:

```
schema filter(l, p):
[ true → Returns ( List < p at Returns(true) ... > ) ]
```

Оператор typeOf. Теперь вместо многоточия осталось вставить конструкцию, которая бы корректно вывела тип, который имеет аргумент лямбды в схеме `p at Returns(true)`.

Однако перед тем, как ввести эту операцию, остановимся на одним маленьком нюансе, на котором мы не заострили внимание в прошлом параграфе. Дело в том, что в момент аннотации `filter`, невозможно знать, какое имя будет у аргумента лямбды, поскольку объявление лямбды будет написано в месте вызова `filter`. Следовательно, мы не можем сослаться на этот аргумент.

Если для сравнения посмотреть на спецификацию `filter` в терминах тео-

рии множеств, то там *вводится* новая переменная x с помощью конструкции $\{x \in S | P(x)\}$. Мы могли бы поступить по аналогии, и сделать еще одну новую конструкцию для введения переменных – нечто в духе `let x in ...`, навеянное языками семейства ML.

Однако в приведенной реализации удалось этого избежать благодаря тому, что синтаксис `Kotlin` позволяет в сигнатуре функции указать именовать аргументы принимаемых на вход лямбд:

```
1 fun <T> filter(s: Collection<T>, p: (arg: T) -> Boolean): List<T> {
2   ...
3 }
```

Теперь мы можем ввести оператор `typeOf`. Выражение `S typeOf V`, где `S` – схема, а `V` – переменная, выдает тип, подтипом которого гарантированно является `V` в схеме `S`, и из таких наиболее частный. Другими словами, оператор `typeOf` в точности соответствует оператору слияния из пункта 3.1.2, использующему в качестве приближенных контекстов классы `DataFlowValue` из анализа приведений типов в пункте 3.3.2.

Таким образом, полная спецификация `filter` выглядит следующим образом:

```
schema filter(l, p):
[ true → Returns ( List <(p at Returns(true)) typeOf arg> ) ]
```

В ней, однако, существует одна досадная, чисто техническая неточность. Дело в том, что при конкретном вызове `filter` вместо `p` будет подставлена лямбда. Эта лямбда может иметь другое мнение на счет того, как должен называться ее первый параметр – соответственно, и ее схема эффектов будет использовать другое имя для этого параметра. Например, вызов `filter(S, { x -> x is String })`, как можно видеть, использует для первого аргумента имя `x`, в то время как с точки зрения схемы этот же самый первый аргумент имеет имя `arg`

В связи с этим, нам необходимо выполнить композицию подстановок – сначала нужно подставить в схему эффектов `filter` все параметры вызова,

а затем в тела лямбд нужно подставить корректные аргументы. Для того, чтобы было проще понять, в какие лямбды нужно подставить какое имя, можно более явно специфицировать вызов, и вместо `p at ...` писать `p(arg) at ...`.

Таким образом, конечный синтаксис записис эффектов для `filter` выглядит следующим образом:

```
schema filter(l, p):
[ true → Returns ( List <(p(arg) at Returns(true)) typeOf arg> ) ]
```

Анализ. Как можно видеть, синтаксис получился не из самых приятных. Кроме того, поддержка подобных функций потребовала значительных усилий со стороны системы эффектов – пришлось ввести несколько новых конструкций, расширить грамматику. В оправдание можно сказать, что эти конструкции не были полностью новыми – они уже использовались внутри системы, и нужно было только выдать им соответствующие операторы в грамматике. Тем не менее, количество усилий определенно выше, чем для того же `Calls`. Почему же так вышло? Не получили ли мы негибкую систему, которая способна формализовать лишь узкий, заранее предопределенный круг понятий?

Ответ состоит в том, что разработанная система эффектов, несмотря на введенные расширения, все же остается системой *эффектов*. Мысль, которую мы пытались формализовать в спецификации `filter`, интуитивно противоречит пониманию эффекта – по сути, она является спецификацией того, как работает `filter`, а не некоторым «побочным» эффектом вычисления.

С другой же стороны, эффект `Calls` превосходно укладывается в эту концепцию – соответственно, и его добавление в систему не потребовало вообще никаких дополнительных действий. К примеру, таким же свойством обладают «классические» эффекты `read` и `write`, часто рассматриваемые в статьях и работах по системам эффектов – их добавление также не потребует практически никаких усилий от системы эффектов (однако могут возникнуть довольно нетривиальные задачи при *использовании* информации об этих эффектах).

В этом, вобщем-то, нет ничего удивительного – чем больше понятий способна формализовать та или иная логическая система, тем более она сложна и тем больше в ней конструкций. Мы изначально отказались от мощных, но сложных систем, реализованных в виде формальных языков спецификаций – как следствие, добавление новых понятий обошлось относительно дорого.

Мы видели, что с точки зрения синтаксиса, теория множеств позволяет формализовать это утверждение ощутимо короче и намного ясней. Однако введение теории множеств *только* для формализации контракта `filter` и других функций, работающих с коллекциями, выглядит неоправданным с практической точки зрения – это серьезный пласт новых понятий, терминов, синтаксиса и работы по реализации всего этого на практике.

В связи с этим, в реализации, приложенной к данной работе, было принято решение остановиться на решении проблемы умных приведений типов в коллекциях с помощью системы эффектов.

Заключение

В данной работе было рассмотрено расширение классических систем эффектов за счет введения условных эффектов. Были предложены правила комбинации эффектов, предусматривающие возможность масштабирования системы путем добавления новых операторов и эффектов.

Полученная система наилучшим образом проявила себя при работе с утверждениями, которые хорошо укладываются в классическое понятие «эффекта» – т.е. некоторое изменение в контексте, производимое подпрограммой. Можно сделать вывод, что в систему, при необходимости, можно будет легко добавить большинство классических эффектов: `read`, `write`, `throws`, и др.

С другой стороны, введение понятий, слабо связанных с побочными эффектами, требует значительно бóльших усилий и добавления новых конструкций и операторов. Так, умные приведения типа в коллекциях требуют формализма, который ближе к теории множеств, и потому укладываются в систему условных эффектов не очень хорошо. Тем не менее, в предложенной системе, технически, имеется возможность выражать и записывать такие понятия.

В качестве примера практической реализации, данная система была реализована в компиляторе языка `Kotlin`. Для учета специфики данного языка (в частности, последовательных вычислений и их частичности), был введен специальный класс эффектов, описывающих исходы вычислений. Знание об этом классе заложено в систему для корректного извлечения эффектов некоторой последовательности вычислений, некоторые из которых завершаются аварийно (с исключением).

За счет добавления этой системы удалось улучшить статический анализ, выполняемый компилятором языка `Kotlin`. В частности, были:

- Поддержаны умные приведения типа, извлекающие информацию из выражений в условных конструкциях и включающих в себя вызовы функций.
- Поддержаны умные приведения типа, опирающиеся на факт (не)успешного

завершения функции (например, в вызовах `assert`).

- Поддержаны умные приведения типа в коллекциях (например, в вызовах вида `Collection<T>.filter`)
- Улучшены существующие в компиляторе диагностики (например, об инициализации переменных) за счет эффектов, описывающих, что функция в ходе своего выполнения вызовет некоторую другую детерминированное число раз.

Также был реализован вывод эффектов из простых выражений.
В качестве возможных направлений развития следует отметить:

- Расширение системы за счет добавления других конструкций логики: кванторы, импликации (в явном виде), предикаты
- Исследование других эффектов: например, связанных с многопоточностью, вводом-выводом, чтением-записью в переменные, и т.д.
- Улучшение системы автоматического вывода эффектов.

Список литературы

[1] Aho Alfred V., Sethi Ravi, Ullman Jeffrey D. Compilers: Principles, Techniques, and Tools. — Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 1986. — ISBN: 0-201-10088-6.

[2] Barth Jeffrey M. A Practical Interprocedural Data Flow Analysis Algorithm // Commun. ACM. — 1978. — sep. — Vol. 21, no. 9. — P. 724–736. — URL: http://doi.acm.org/10.1145/359588.359596.

[3] Blais E., Tan L. Y. Approximating Boolean Functions with Depth-2 Circuits // 2013 IEEE Conference on Computational Complexity. — 2013. — June. — P. 74–85.

[4] CASL: the Common Algebraic Specification Language / Egidio Astesiano, Michel Bidoit, Hélène Kirchner et al. // Theoretical Computer Science. — 2002. — Vol. 286, no. 2. — P. 153 – 196. — URL: http://www.sciencedirect.com/science/article/pii/S0304397501003681.

[5] Checker Framework // Checker Framework online reference. — 2017. — URL: https://checkerframework.org/ (online; accessed: 08.05.2017).

[6] Equality - Kotlin Programming Language // Kotlin Online Reference. — URL: https://kotlinlang.org/docs/reference/equality.html (online; accessed: 2017-05-24).

[7] Ernst Michael D. Type Annotations specification (JSR 308). — http://types.cs.washington.edu/jsr308/. — 2008. — September 12,.

[8] Experience with Z developing a control program for a radiation therapy machine / Jonathan Jacky, Jonathan Unger, Michael Patrick et al. // ZUM '97: The Z Formal Specification Notation: 10th International Conference of Z Users Reading, UK, April 3–4, 1997 Proceedings / Ed. by Jonathan P. Bowen, Michael G. Hinchey, David Till. — Berlin, Heidelberg : Springer Berlin Heidelberg, 1997. — P. 317–328. — ISBN: 978-3-540-68490-9. — URL: http://dx.doi.org/10.1007/BFb0027295.

[9] Greenhouse Aaron, Boyland John. An Object-Oriented Effects System // Proceedings of the 13th European Conference on Object-Oriented Programming. — ECOOP '99. — London, UK, UK : Springer-Verlag, 1999. — P. 205–229. — URL: http://dl.acm.org/citation.cfm?id=646156.679836.

[10] Guttag John V., Horning James J. Larch: Languages and Tools for Formal Specification. — New York, NY, USA : Springer-Verlag New York, Inc., 1993. — ISBN: 0-387-94006-5.

[11] Hoare C. A. R. An Axiomatic Basis for Computer Programming // Commun. ACM. — 1969. — Vol. 12, no. 10. — P. 576–580. — URL: http://doi.acm.org/10.1145/363235.363259.

[12] Inferring Method Effect Summaries for Nested Heap Regions / M. Vakilian, D. Dig, R. Bocchino et al. // 2009 IEEE/ACM International Conference on Automated Software Engineering. — 2009. — Nov. — P. 421–432.

[13] Jain T. K., Kushwaha D. S., Misra A. K. Optimization of the Quine-McCluskey Method for the Minimization of the Boolean Expressions // Fourth International Conference on Autonomic and Autonomous Systems (ICAS'08). — 2008. — March. — P. 165–168.

[14] Jouvelot P., Gifford D. K. The FX-87 Interpreter // Proceedings. 1988 International Conference on Computer Languages. — 1988. — Oct. — P. 65–72.

[15] Lucassen J. M., Gifford D. K. Polymorphic Effect Systems // Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. — POPL '88. — New York, NY, USA : ACM, 1988. — P. 47–57. — URL: http://doi.acm.org/10.1145/73560.73564.

[16] McCluskey Edward J. Minimization of Boolean functions // Bell Labs Technical Journal. — 1956. — Vol. 35, no. 6. — P. 1417–1444.

[17] Meyer Bertrand. Applying "Design by Contract" // Computer. — 1992. — oct. — Vol. 25, no. 10. — P. 40–51. — URL: http://dx.doi.org/10.1109/2.161279.

[18] Muchnick Steven S. Advanced Compiler Design and Implementation. — San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 1997. — ISBN: 1-55860-320-4.

[19] Nielson Flemming, Nielson Hanne Riis. Type and Effect Systems // Correct System Design: Recent Insights and Advances / Ed. by Ernst-Rüdiger Olderog, Bernhard Steffen. — Berlin, Heidelberg : Springer Berlin Heidelberg, 1999. — P. 114–136. — ISBN: 978-3-540-48092-1. — URL: http://dx.doi.org/10.1007/3-540-48092-7_6.

[20] Pierce Benjamin C. Types and Programming Languages. — 1st edition. — The MIT Press, 2002. — ISBN: 0262162091, 9780262162098.

[21] Sagiv Mooly, Reps Thomas, Horwitz Susan. Precise interprocedural dataflow analysis with applications to constant propagation // Theoretical Computer Science. — 1996. — Vol. 167, no. 1. — P. 131 – 170. — URL: http://www.sciencedirect.com/science/article/pii/0304397596000722.

[22] Schwartz J.T., Society A.M. Mathematical Aspects of Computer Science. American Mathematical Society. Proceedings of Symposia in Applied Mathematics. — American Mathematical Soc., 1967. — ISBN: 9780821867280. — URL: https://books.google.ru/books?id=ynigSICJflYC.

[23] Sharir Micha, Pnueli Amir. Two approaches to interprocedural data flow analysis. — 1978.

[24] Spivey J. M. The Z Notation: A Reference Manual. — Hertfordshire, UK, UK : Prentice Hall International (UK) Ltd., 1992. — ISBN: 0-13-978529-9.

[25] Type Checks and Casts - Kotlin Programming Language // Kotlin Online Reference. — URL: https://kotlinlang.org/docs/reference/typecasts.html (online; accessed: 2017-05-24).

[26] Types for Atomicity: Static Checking and Inference for Java / Cormac Flanagan, Stephen N. Freund, Marina Lifshin, Shaz Qadeer // ACM Trans. Program. Lang. Syst. — 2008. — Vol. 30, no. 4. — P. 20:1–20:53. — URL: http://doi.acm.org/10.1145/1377492.1377495.

[27] Weihl William E. Interprocedural Data Flow Analysis in the Presence of Pointers, Procedure Variables, and Label Variables // Proceedings of the 7th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. — POPL '80. — New York, NY, USA : ACM, 1980. — P. 83–94. — URL: http://doi.acm.org/10.1145/567446.567455.

Приложение А. Грамматика

```
grammar EffectSystem;


// Entry-point
effectSchema
    : EOF
    | clause (SEMI clause)*
    ;

clause
    : expression '->' effectsList
    ;


// Expressions

expression
    : conjunction (disjunctionOperator conjunction)*
    ;

conjunction
    : equalityComparison
      (conjunctionOperator equalityComparison)*
    ;

equalityComparison
    : comparison (equalityOperator comparison)*
    ;

comparison
    : namedInfix (comparisonOperator namedInfix)*
    ;

namedInfix
    : additiveExpression (inOperation additiveExpression)*
    | additiveExpression atOperation effect
    | additiveExpression (isOperation type)?
    ;

additiveExpression
    : multiplicativeExpression
    (additiveOperator multiplicativeExpression)*
    ;

multiplicativeExpression
    : prefixUnaryExpression
      (multiplicativeOperator prefixUnaryExpression)*
    ;

prefixUnaryExpression
    : prefixUnaryOperation* postfixUnaryExpression
    ;

postfixUnaryExpression
    : atomicExpression postfixUnaryOperation*
    ;

atomicExpression
    : '(' expression ')'
    | literalConstant
    | SimpleName
    ;

disjunctionOperator
    : '||'
    ;

conjunctionOperator
    : '&&'
    ;

equalityOperator
    : EQEQ
    | EXCLEQ
    ;

comparisonOperator
    : LT | GT | LEQ | GEQ
    ;

additiveOperator
    : PLUS | MINUS
    ;

multiplicativeOperator
    : MUL | DIV | PERC
    ;

prefixUnaryOperation
    : MINUS | PLUS
    | MINUSMINUS | PLUSPLUS
    | NOT
    ;

postfixUnaryOperation
    : PLUSPLUS | MINUSMINUS | EXCLEXCL
    | callSuffix
    ;

callSuffix
    : '(' (expression (',' expression)*)? ')'
    ;

inOperation
    : 'in' | '!in'
    ;

isOperation
    : 'is' | '!is'
    ;

atOperation
    : 'at'
    ;


// Effects

effectsList
    : effect (',' effect)*
    ;

effect
    : throwsEffect
```

```
    | returnsEffect
    | callsEffect
    | hintsEffect
    ;

throwsEffect
    : 'Throws' type
    ;

returnsEffect
    : 'Returns' '(' (expression | UnknownLiteral ) ')'
    ;

callsEffect
    : 'Calls' '(' callsRecord (';' callsRecord)* ')'
    ;

callsRecord
    : SimpleName IntegerLiteral;

hintsEffect
    : 'Hints' '(' SimpleName ',' typeExpression ')'
    ;

literalConstant
    : BooleanLiteral
    | IntegerLiteral
    | StringLiteral
    | NullLiteral
    | UnitLiteral
    ;

typeExpression
    : type
    | typeOfOperator
    ;

type
    : SimpleName typeParametersList?
    ;

typeOfOperator
    : expression 'typeOf' SimpleName
    ;

typeParametersList
    : '<' typeExpression (',' typeExpression)* '>'
    ;

BooleanLiteral
    : 'true'
    | 'false'
    ;

NullLiteral : 'null';

UnknownLiteral : 'unknown';

UnitLiteral : 'unit';


// String literals
StringLiteral
    :   '"' StringCharacters? '"'
    ;

fragment
StringCharacters
    :   StringCharacter+
    ;

fragment
StringCharacter
    :   ~["\\]
    |   EscapeSequence
    ;

fragment
EscapeSequence
    :   '\\' [btnfr"'\\]
    |   OctalEscape
    |   UnicodeEscape
    ;

fragment
OctalEscape
    :   '\\' OctalDigit
    |   '\\' OctalDigit OctalDigit
    |   '\\' ZeroToThree OctalDigit OctalDigit
    ;

fragment
UnicodeEscape
    :   '\\' 'u' HexDigit HexDigit HexDigit HexDigit
    ;

fragment
ZeroToThree
    :   [0-3]
    ;

// Numeric literals

//TODO add hex/binary/octal integers
IntegerLiteral
    :   DecimalIntegerLiteral
    ;

fragment
DecimalIntegerLiteral
    :   DecimalNumeral IntegerTypeSuffix?
    ;

fragment
IntegerTypeSuffix
    :   [lL]
    ;

fragment
DecimalNumeral
    :   '0'
    |   NonZeroDigit (Digits? | Underscores Digits)
    ;

fragment
Digits
    :   Digit (DigitOrUnderscore* Digit)?
    ;

fragment
Digit
    :   '0'
    |   NonZeroDigit
    ;
```

```
259
260     fragment
261     NonZeroDigit
262         :   [1-9]
263         ;
264
265     fragment
266     DigitOrUnderscore
267         :   Digit
268         |   '_'
269         ;
270
271     fragment
272     Underscores
273         :   '_'+
274         ;
275
276     fragment
277     OctalDigit
278         :   [0-7]
279         ;
280
281     fragment
282     HexDigit
283         :   [0-9a-fA-F]
284         ;
285
286     // Identifiers
287
288     SimpleName
289         :   JavaLetter JavaLetterOrDigit*
290         ;
291
292     fragment
293     JavaLetter
294         :   [a-zA-Z$_]
295         |   ~[\u0000-\u007F\uD800-\uDBFF]
296         |   [\uD800-\uDBFF] [\uDC00-\uDFFF]
297         ;
298
299     fragment
300     JavaLetterOrDigit
301         :   [a-zA-Z0-9$_] // these are the "java letters or digits" below 0x7F
302         |   // covers all characters above 0x7F which are not a surrogate
303             ~[\u0000-\u007F\uD800-\uDBFF]
304         |   // covers UTF-16 surrogate pairs encodings for U+10000 to U+10FFFF
305             [\uD800-\uDBFF] [\uDC00-\uDFFF]
306         ;
307
308
309     WS  :  [ \t\r\n\u000C]+ -> skip
310         ;
311
312     EOL : '\r'? '\n';
313
314     SEMI : ';';
315
316     LT : '<';
317     GT : '>';
318     LEQ : '<=';
319     GEQ : '>=';
320     PLUS : '+';
321     MINUS : '-';
322     MUL : '*';
323     DIV : '/';
324     PERC : '%';
325     PLUSPLUS : '++';
326     MINUSMINUS : '--';
327     NOT : '!';
328     EXCLEXCL : '!!';
329     EQEQ : '==';
330     EXCLEQ : '!=';
```