Санкт-Петербургский государственный университет

Математическое обеспечение и администрирование информационных систем

Системное программирование

Сокольвяк Сергей Дмитриевич

Предметно-ориентированные аннотации для типов в Java

Выпускная квалификационная работа

Научный руководитель:
доц. кафедры СП, к. ф.-м. н. К. Ю. Романовский

Научный консультант:
старший преподаватель кафедры СП Я. А. Кириленко

Рецензент:
программист ООО "ИнтеллиДжей Лабс", к. ф.-м. н. Д. А. Березун

Санкт-Петербург
2020

SAINT PETERSBURG STATE UNIVERSITY

Software and Administration of Information Systems
System Programming

Sergey Sokolvyak

Domain-specific annotations for Java types

Bachelor's Thesis

Scientific supervisor:
Associate Professor Konstantin Romanovsky

Scientific advisor:
Senior Lecturer Iakov Kirilenko

Reviewer:
Software engineer LCC "IntelliJ Labs" Daniil Berezun

Saint Petersburg
2020

Оглавление

Введение **5**

Благодарности **7**

1. Постановка задачи **8**

2. Обзор **9**

2.1. Возможные подходы к решению проблемы 9

2.1.1. Изменение архитектуры: объявление новых типов с помощью Java классов 9

2.1.2. Смена языка программирования 10

2.1.3. Использование библиотек единиц измерения . . . 11

2.1.4. Использование аннотаций для разметки и анализа исходного кода программы 12

2.2. Annotation Processing Tool. Инструментальное средство для анализа аннотаций 13

2.3. Похожее решение. Checker Framework 14

2.4. Выводы 15

3. Описание алгоритма анализа исходного кода **16**

3.1. Правила вывода уточняющих типов 17

3.1.1. Правила вывода типов для литералов и идентификаторов 17

3.1.2. Правила вывода типов для вызовов методов и конструкторов 17

3.1.3. Правила вывода типов для объявлений локальных переменных и полей 18

3.1.4. Правила вывода типов для выражений с бинарными и унарными операторами 18

3.1.5. Правила вывода типов для выражений с обращением к массиву и тернарным оператором 19

3.1.6. Правила вывода типов объявлений методов и конструкторов 19

4. Архитектура инструментального средства и особенности реализации 20

4.1. Архитектура системы 20
4.2. Статические проверки соответствия размеченных типов 22
4.3. Создание и использование иерархии подтипов 23
4.4. Использование типов в операторах 24
4.5. Вывод сообщений об ошибках 25

5. Апробация реализованного инструмента 26

Заключение 28

Список литературы 29

Введение

При разработке современных промышленных приложений зачастую возникает множество разнообразных и трудноуловимых ошибок, которые могут приводить к некорректному поведению программы во время исполнения. Эффективным языковым средством, позволяющим избегать многих ошибок, является система типов языка.

Основная роль системы типов любого языка программирования, как формального метода проверки корректности программ, заключается в сокращении числа ошибок, возникающих в этих программах [14]. Это возможно благодаря классификации выражений языка по разновидностям вычисляемых ими значений. Классы в такой классификации называются типами языка программирования.

Несмотря на высокую эффективность, системы типов гарантируют отсутствие только определенных видов ошибок [6]. Например, большинство систем типов способны статически проверить, что в качестве фактического параметра метода, принимающего целое число, действительно используется целочисленное значение. Однако системы типов не способны гарантировать, чтобы при вызове метода пользователь в правильном порядке перечислял аргументы, которые принадлежат одному и тому же типу.

В языке программирования Java подобные ошибки особенно часто возникают при использовании примитивных типов. К сожалению в промышленном коде подобные ошибки приводят к нежелательным последствиям, в том числе к убыткам. Чтобы избегать таких ошибок, можно размечать примитивные типы дополнительной информацией, а затем с помощью статических проверок, использующих эту информацию, находить подобные ошибки в коде на этапе компиляции программы.

При использовании подобного подхода следует помнить, что статические проверки могут оказаться слишком строгими и отвергать корректно работающие программы. Например, программа, которая взаимодействует с реляционной базой данных, может использовать в своих методах идентификаторы пользователей из разных таблиц и размечен-

ных по-разному. Статические проверки запретят использовать идентификатор одного размеченного типа в качестве другого. Однако идентификаторы пользователей из первой таблицы можно безопасно использовать там, где ожидается идентификатор из второй, если первая таблица содержит дополнительную информацию о всех пользователях из второй таблицы. В таких случаях полезным оказывается возможность определять один тип вложенным [14] в другой тип или один тип подтипом другого типа.

Некоторые виды некорректного поведения программ может быть трудно обнаружить. Например, программа, которая взаимодействует с реляционной базой данных, может использовать в качестве идентификатора целочисленную переменную. Система типов языка разрешит использование этой переменной в арифметических операциях, однако с точки зрения смысла такое поведение редко оказывается корректным.

Возможно подобные проблемы не возникали бы при использовании более современных языков программирования, таких как Scala [1] или Kotlin [2]. Однако переход на другой язык программирования в промышленном проекте не всегда возможен, так как сопровождается издержками. Кроме того, Java является одним из самым популярных языков [3], который распространен повсеместно.

Таким образом, возникает необходимость в инструменте, который позволит избегать подобных ошибок в программах на языке Java.

[1]https://www.scala-lang.org

[2]https://kotlinlang.org

[3]https://www.tiobe.com/tiobe-index/java

Благодарности

Я хотел бы поблагодарить Якова Александровича Кириленко за значительный вклад в развитие данной работы и помощь в написании этого текста. Я также хотел бы поблагодарить Филиппа Долголева, Антона Плотникова и компанию DSX Technologies за контроль развития проекта, предложенные идеи и предоставление ценной обратной связи на протяжении всего процесса. Особая благодарность Даниилу Андреевичу Березуну за рецензирование этой работы и конструктивные замечания.

1. Постановка задачи

Цель данной работы – уменьшить количество ошибок, возникающих при использовании примитивных типов для обозначения сущностей из разных предметных областей в языке Java. Для достижения этой цели были поставлены следующие задачи.

1. Исследовать предметную область: рассмотреть возможные подходы к решению возникшей проблемы, а также выбрать подходящий в контексте конкретной промышленной задачи способ решения.

2. В контексте выбранного подхода к решению проблемы, придумать алгоритм анализа программы, позволяющий находить упомянутые ошибки и указывать на них пользователю.

3. На основе разработанного алгоритма, реализовать инструментальное средство, позволяющее

   (a) размечать примитивные типы дополнительной информацией;

   (b) статически проверять корректность использования размеченных типов;

   (c) создавать иерархию подтипов на размеченных типах;

   (d) ограничивать использование размеченных типов в выбранных бинарных и унарных операторах.

4. Провести апробацию и тестирование реализованного инструментального средства.

2. Обзор

В данном обзоре рассматриваются способы достижения поставленной цели в контексте упомянутой проблемы, возникшей в промышленном проекте компании DSX. Кроме того, в рамках данного обзора описываются схожая система и технология, необходимая для реализации выбранного способа решения.

2.1. Возможные подходы к решению проблемы

Для достижения поставленной цели было принято решение рассмотреть широкий спектр инструментальных подходов к решению упомянутых задач.

2.1.1. Изменение архитектуры: объявление новых типов с помощью Java классов

Создание новых типов с помощью Java классов действительно позволяет избежать многих ошибок в программах. Например, правильный порядок перечисления аргументов при вызове метода можно гарантировать, если создать новый класс для каждого из одинаково типизированных аргументов этого метода, так как корректное использование классов контролируется системой типов Java. Ввиду того, что Java является объектно-ориентированным языком, классы в языке Java поддерживают наследование. Это позволяет строить иерархию подтипов на таких типах. А возможность объявления методов в классах позволяет определять аналоги необходимым арифметическим операциям над значениями таких типов.

Существует несколько способов использования Java классов для различия однотипных, но концептуально разных значений. Одним из вариантов является использование классов в качестве оберток. Такой подход имеет несколько недостатков, таких как необходимость в постоянной упаковке и распаковке значений при использовании методов сторонних библиотек, а также снижение производительность при использовании

классов вместо примитивных типов. Кроме того, операции проверки эквивалентности и идентичности могут возвращать неправильные результаты, что может приводить к возникновению ошибок. А отсутствие возможности перегружать операторы вынуждает определять аналогичные методы, которые усложняют понимание исходного кода программы.

Другим вариантом использования классов является объявление новых производных классов для концептуально разных значений. Однако примитивные типы являются завершенными в Java и не поддерживают объявление наследников. Несколько других существенных недостатков такого подхода представлены в статье [7].

2.1.2. Смена языка программирования

Существуют языки программирования, системы типов которых предоставляют более эффективные способы избегать ошибок, связанных с путаницей однотипных переменных, нежели система типов Java. Однако миграция проекта на другой язык программирования зачастую приводит к большим затратам, связанным с обучением программистов, переписыванием исходного кода программы, а также нарушениями в графике работ.

При миграции с Java логичным выбором альтернативы может оказаться язык программирования Kotlin [4]. Его полная совместимость с Java, а также поддержка автоматической миграции кода с помощью IntelliJ IDEA [5] позволяют уменьшить сопутствующие затраты. Кроме того, такие преимущества, как поддержка перегрузки операторов и встроенные классы[6], действительно позволяют эффективно избегать ошибок, связанных с путаницей однотипных переменных. В то же время встроенные классы не могут участвовать в отношении наследования, а значит на них нельзя построить иерархию подтипов. Кроме того, в реальных проектах при переходе с Java на Kotlin могут возникнуть

[4] https://kotlinlang.org
[5] https://www.jetbrains.com/ru-ru/idea/
[6] https://kotlinlang.org/docs/reference/inline-classes.html

непредвиденные трудности [7], особенно если в них используются сторонние библиотеки, основанные на аннотациях, например, Dagger [8] или Lombok [9].

2.1.3. Использование библиотек единиц измерения

Единицы измерения позволяют уточнять численные значения размерностью. Таким образом, библиотеки, расширяющие язык программирования Java единицами измерения и проверками их соответствия, помогают различать однотипные, но концептуально разные переменные. В качестве примера для рассмотрения была выбрана система, описанная в Java Specification Request (JSR) 385 [10], так как эта система была признана Java Community Process лучшим JSR 2019 года. Решение, предложенное в JSR 385, определяет набор констант перечислений, соответствующих единицам измерения СИ, методы для работы с ними, а также статические проверки корректного использования единиц измерения (рис. 1). Кроме того, это решение предоставляет возможность

```
public void m() {
   final Unit<Length> METRE = Units.METRE;
   final Unit<Length> FOOT =
            METRE.multiply(3048).divide(10000);

   Quantity<Length> x =
       Quantities.getQuantity(10.0, METRE);
   Quantity<Length> y =
       Quantities.getQuantity(10.0, FOOT);

   Quantity<Length> sum1 = x.add(y);
   Quantity<Length> sum2 = y.add(x);
```

Рис. 1: Пример использования JSR385 [12]

объявлять собственные единицы измерения, определяя соответствую-

[7] https://habr.com/ru/post/421871
[8] https://dagger.dev
[9] https://projectlombok.org

щие константы перечислений.

На первый взгляд такая библиотека может помочь достичь поставленной цели. Однако подобное решение не позволяет создавать иерархии подтипов на единицах измерения, а также ограничивать их использование в арифметических операциях. Кроме того, создание и использование переменных, параметризованных единицами измерения, требует переписывать исходный код программы в соответствии с новыми типами. В дополнение вышесказанному, библиотеки единиц измерения созданы для указания размерности численных величин. В контексте поставленных нами задач величины, типы которых необходимо размечать, могут использоваться не только для указания количества каких-либо единиц, но и в качестве уникальных номеров каких-либо объектов из предметной области, или иметь строковый тип. Таким образом, задачи, которые решают единицы измерения, отличаются от поставленных в контексте данной работы задач.

2.1.4. Использование аннотаций для разметки и анализа исходного кода программы

Java поддерживает языковое средство, позволяющее встраивать в исходный код программы дополнительную информацию [1]. Информация, о которой идет речь, называется аннотацией. Аннотации не оказывают влияние на работу программы, оставляя семантику программ неизменной. Эта информация может быть проанализирована обработчиками аннотаций и использована компилятором для вывода дополнительных ошибок или предупреждений.

Идея данного подхода заключается в использовании аннотаций для разметки типов в исходном коде программы на языке Java. Анализ таких аннотаций с помощью инструментальных средств позволяет избегать ошибок, связанных с путаницей однотипных переменных.

Как уже было сказано, аннотации не оказывают влияния на работу программы. Кроме того, аннотации можно анализировать на этапе компиляции программы. Это означает, что использование аннотаций не оказывает влияния на производительность прикладных программы.

Это особенно важно для систем, которые выполняют большой объем вычислений в реальном времени или проводят большое число платежных транзакции, например, биржи. Исходя из рассмотренных подходов к решению возникшей проблемы, было принято решение использовать именно этот подход, так как он позволит достичь всех поставленных задач.

2.2. Annotation Processing Tool. Инструментальное средство для анализа аннотаций

Annotation Processing Tool [3] — это мощное инструментальное средство, позволяющее создавать плагины, расширяющие возможности компилятора Java с помощью сканирования и анализа аннотаций в исходном коде программы. Annotation Processing является одним из этапов процесса компиляции программы на языке Java, так же как и построение дерева разбора или удаление синтаксического сахара [5].

Анализ или обработка аннотаций происходит за несколько раундов. Каждый раунд начинается с того, что компилятор сканирует файлы с исходным кодом на предмет аннотаций и выбирает обработчики аннотаций, подходящие для этих аннотаций. Каждый обработчик в свою очередь анализирует соответствующие файлы с исходным кодом. Раунд завершается, когда все файлы с исходным кодом, содержащие аннотации, будут проанализированы. Если на протяжении текущего раунда процессор аннотаций создал какие-либо файлы с исходным кодом, то эти файлы подаются на вход следующему раунду. Этот процесс продолжается до тех пор, пока на вход очередному раунду не будет передан пустой набор файлов с исходным кодом [13].

Как уже было сказано, анализ аннотаций производится соответствующим обработчиком или процессором аннотаций. Для того чтобы определить пользовательский процессор аннотаций, необходимо реализовать интерфейс Processor, а в методе `process` определить логику обработки аннотаций. Для указания компилятору, какие аннотации способен обработать конкретный процессор аннотаций, необходи-

мо реализовать метод getSupportedAnnotationTypes или использовать аннотацию SupportedAnnotationTypes в объявлении класса, реализующего процессор аннотаций. Для того чтобы компилятор использовал процессор аннотаций, его необходимо зарегистрировать, указав в файле `META-INF.services/javax.annotation.processing.Processor` полное название пользовательского процессора аннотаций.

2.3. Похожее решение. Checker Framework

Используя возможности Annotation Processing Tool, программисты из Университета Вашингтона разработали систему Checker Framework.

Checker Framework [2] — это инструмент, расширяющий систему типов языка Java с помощью аннотаций. Благодаря этому разработчики программного обеспечения могут обнаруживать и предотвращать различные ошибки в программах на Java. Checker Framework состоит из более чем двадцати различных обработчиков аннотаций, которые позволяют находить различные некорректные типы поведения, а также проверять их отсутствие.

Одним из таких обработчиков является Subtyping Checker. Subtyping Checker проверяет только правильность использования размеченных типов, анализируя аннотации, указанные пользователем в командной строке при компиляции прикладной программы на Java. Таким образом, пользователи могут создавать простую проверку типов без написания какого-либо кода, выходящего за рамки определений аннотаций для типов.

Одним из недостатков Checker Framework является его громоздкость. Этот инструмент способен обнаруживать большое множество различных ошибок, в том числе ошибки использования пустых ссылок или несоответствия строки заданному шаблону. Эта функциональность выходит далеко за рамки нашей работы и не имеет ничего общего с ошибками, возникающими из-за путаницы однотипных, но концептуально разных переменных или параметров. Таким образом, эта функциональность оказывается бесполезной при решении поставленных задач,

однако может помешать расширению системы, усложнить использование и интеграцию, а также значительно увеличить время компиляции прикладных программ. С другой стороны, Subtyping Checker, который является небольшой частью этого инструмента, не удовлетворяет всем поставленным задачам. В частности, нельзя запрещать или ограничивать использование типов в арифметических операциях. В дополнение к вышесказанному на момент начала нашей работы Checker Framework поддерживал только JDK6.

2.4. Выводы

Исходя из выбранного подхода к решению, а также недостатков существующей системы, возникает необходимость в разработке алгоритма для анализа размеченных аннотациями типов в исходном коде программы. Кроме того, на основе данного алгоритма необходимо реализовать инструментальное средство, удовлетворяющее поставленным задачам. В частности, позволяющее создавать и использовать иерархию подтипов на размеченных типах, а также ограничивать использование размеченных типов в бинарных и унарных операторах.

3. Описание алгоритма анализа исходного кода

Алгоритм анализа исходного кода программы проверяет только корректное использование типов, указанных в аннотациях. В дальнейшем будем называть такие типы уточняющими. Корректность использования примитивных и пользовательских типов проверяет сама система типов Java.

Алгоритм получает на вход абстрактное синтаксическое дерево исходного кода программы. Работа алгоритма заключается в последовательном выводе уточняющих типов узлов абстрактного синтаксического дерева, начиная от идентификаторов и литералов, вызовов конструкторов и методов, объявлений полей и локальных переменных и заканчивая методами классов, объявленных в исходном коде. Если в какой-то момент работы алгоритма невозможно вывести уточняющий тип синтаксического терма, то работа алгоритма останавливается и возвращается соответствующая ошибка компиляции. Синтаксис языка Java в нотации БНФ [10], описывающий синтаксические термы языка, представлен в [9].

Для того чтобы алгоритм не отвергал программы, в которых используются подтипы уточняющих типов, используется принцип безопасной подставки [14]. Для применения принципа безопасной подстановки необходимо проверять, является ли один тип подтипом другого типа. Данная задача широко исследована в статье авторов из Университета Пенсильвании и Microsoft Research Cambridge [11]. Автор из Кентского университета в статье [8] представил доказательство неразрешимости этой задачи в общем случаи при использовании параметризованных типов [11]. В рамках данного алгоритма в качестве уточняющих типов не используются параметризованные типы, и для каждого уточняющего типа можно определять единственный непосредственный объемлющий тип. Кроме того, запрещено строить иерархии подтипов, в которых последовательное применение правила $\frac{S<:U \quad U<:T}{S<:T}$ (правило транзитивно-

[10] http://matt.might.net/articles/grammars-bnf-ebnf

сти) для уточняющих типов позволяет получить $S <: S$. Запись $T <: S$ означает, что тип `T` является подтипом типа `S`. Кроме того, если при объявлении типа не указан его непосредственный объемлющий тип, то таким типом неявно становится UnknownType. Благодаря упомянутым ограничениям иерархия подтипов на уточняющих типах представляется деревом с корнем UnknownType, а задача проверки вложенности типов является разрешимой [11].

3.1. Правила вывода уточняющих типов

В данном разделе приводятся правила вывода уточняющих типов для синтаксических термов, участвующих в анализе. Если при выводе типа какого-либо из перечисленных выражений нельзя применить соответствующее правило вывода типа, то работа алгоритма останавливается и возвращается соответствующая ошибка компиляции.

3.1.1. Правила вывода типов для литералов и идентификаторов

Ввиду того, что алгоритм анализирует только уточняющие типы, указанные в аннотации, для не размеченных пользователем синтаксических термов определён специальный тип RawType. Таким образом, литералы всегда имеют уточняющий тип RawType. Идентификаторам присваивается уточняющий тип, соответствующий локальным переменным, параметрам методов или полям, на которые они могут указывать.

3.1.2. Правила вывода типов для вызовов методов и конструкторов

Для вывода уточняющего типа выражений, являющихся вызовом методов или конструкторов, используется следующее правило. Если уточняющие типы аргументов являются подтипами уточняющих типов соответствующих формальных параметров, то вызов метода или конструктора получает тип, выведенный для объявления соответствующего метода или конструктора.

3.1.3. Правила вывода типов для объявлений локальных переменных и полей

Для вывода уточняющего типа выражений, являющихся объявлениями полей используется следующее правило. Если тип поля аннотирован, то выражение имеет уточняющий тип, указанный в аннотации. Если при объявлении полю присваивается значение, то выражение типизируется алгоритмом только в том случае, если уточняющий тип инициализирующего выражения является подтипом уточняющего типа, указанного в аннотации, или является типом RawType. Это позволяет инициализировать переменные и поля значениями не размеченных типов, например, литералами. Если поле не аннотировано, тип инициализатора должен быть типом RawType. Подобным образом анализируется объявление локальной переменной. Отличие заключается в том, что в случае отсутствия аннотации объявление переменной получает уточняющий тип инициализирующего выражения, то есть происходит вывод типа локальной переменной.

3.1.4. Правила вывода типов для выражений с бинарными и унарными операторами

Для вывода уточняющего типа выражений с бинарным оператором используется следующее правило. Сначала алгоритм выводит уточняющие типы левого и правого операндов. Если тип левого операнда является подтипом правого, то алгоритм проверяет применимость бинарной операции к данному типу. Если операция применима, то все выражение получает уточняющий тип правого операнда. Аналогично выполняется анализ в случае, если тип правого операнда является подтипом левого. Использование такого подхода при вычислении типа выражений с бинарным оператором мотивировано аналогичным механизмом автоматического продвижения типов в Java [15]. Если к выражению применяется унарный оператор, то уточняющий тип выражения не меняется, однако алгоритм проверяет, разрешено ли применять данный оператор к этому уточняющему типу. Для операций присваивания уточняющий

тип инициализирующего выражения должен быть подтипом уточняющего типа идентификатора, которому присваивается значение. В таком случае типом выражения с присваиванием является уточняющий тип идентификатора. При анализе выражений с составным оператором присваивания, например, `+=` дополнительно проверяется, разрешено ли применять соответствующий оператор к уточняющему типу идентификатора. Например, для составного оператора присваивания `+=` проверяется, разрешено ли применять оператор `+` к уточняющему типу идентификатора в левой части выражения.

3.1.5. Правила вывода типов для выражений с обращением к массиву и тернарным оператором

Выражение, которое соответствует обращению к массиву, получает уточняющий тип, указанный в аннотации при объявлении типа массива. Если соответствующая аннотация не указана, то выражение получает тип `RawType`. Для вывода типа тернарного оператора, алгоритм вычисляет уточняющие типы выражений в обеих ветвях. Среди их общих объемлющих типов вычисляется такой, что он является подтипом любого другого общего объемлющего типа для обеих ветвей. Такой тип всегда существует, так как иерархия подтипов на уточняющих типах представляется деревом.

3.1.6. Правила вывода типов объявлений методов и конструкторов

При выводе уточняющего типа выражений, соответствующих объявлениям методов или конструкторов, алгоритм вычисляет тип всех выражений в теле метода или конструктора. Кроме того, для метода дополнительно проверяется, что типы выражений `return` являются подтипами уточняющего типа, указанного в аннотации к типу возвращаемого значения. Если данные условия выполнены, то объявление метода получает тип указанный в аннотации, а в случае отсутствия аннотации тип RawType. Объявление конструктора получает тип RawType.

4. Архитектура инструментального средства и особенности реализации

В рамках данной главы описывается архитектура, а также особенности реализации инструментального средства, разработанного на основе описанного алгоритма анализа исходного кода программы.

4.1. Архитектура системы

Для реализации упомянутого выше алгоритма анализа была разработана архитектура инструментального средства, представленная на рис 2.

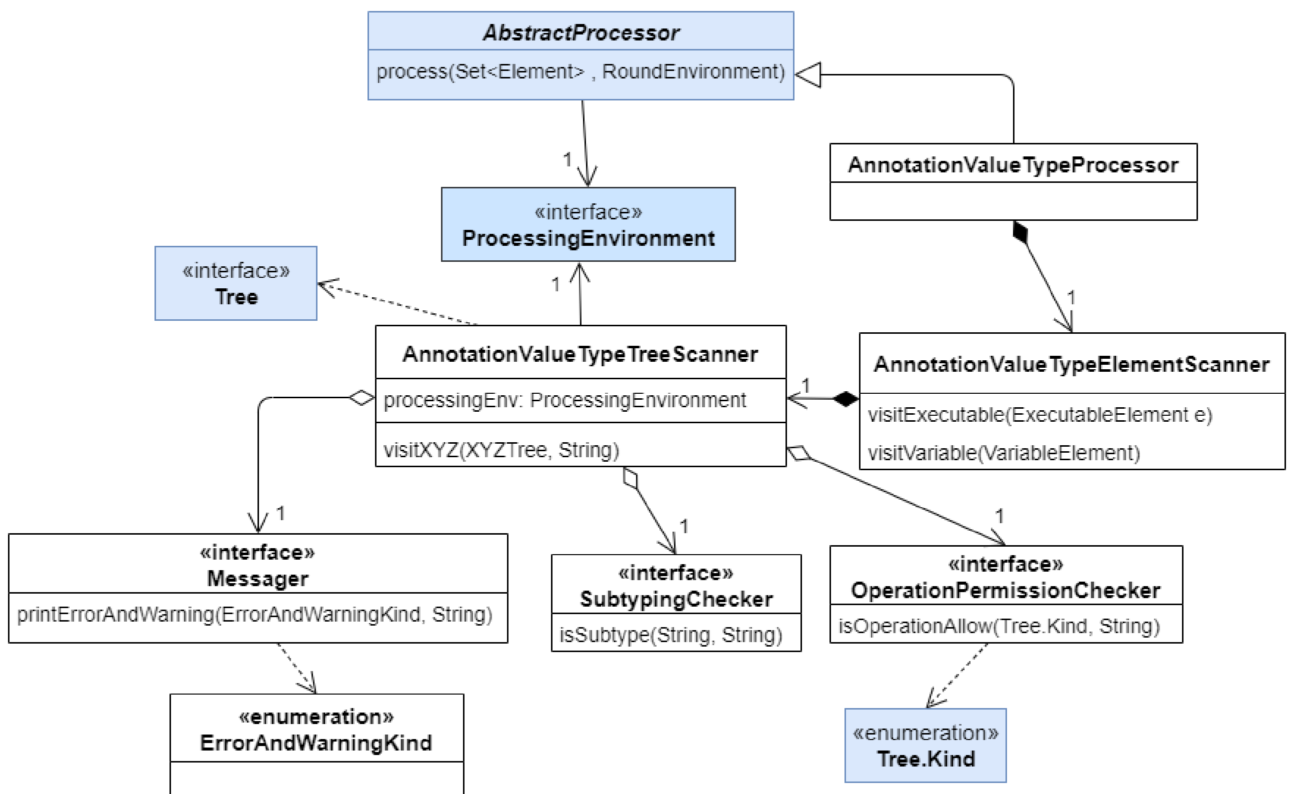

Рис. 2: Основные классы системы, синим выделены классы модулей java.compile и jdk.compiler

Разработанная система представляет собой плагин, подключаемый к компилятору Java в качестве процессора аннотаций. Для этого служит класс AnnotationValueTypeProcessor. Такой подход позволяет по необходимости легко подключать систему к любому компилятору Java.

Для этого достаточно указать JAR-файл библиотеки в качестве процессора аннотаций в настройках выбранной IDE, системы сборки или передать в качестве параметра запуска, если компиляция происходит с помощью утилит командной строки.

Для реализации упомянутого алгоритма анализа исходного кода программы было принято решение воспользоваться интерфейсами из пакетов `com.sun.source.tree` [13] и `javax.lang.model.element` [13]. Эти интерфейсы, предоставляют доступ к абстрактному синтаксическому дереву, построенному компилятором Java. При анализе абстрактного синтаксического дерева узлы, представляющие бинарные операторы и идентификаторы переменных, имеют различные представления и должны анализироваться по-разному. Для того чтобы не усложнять интерфейсы узлов соответствующими методами анализа, а также иметь возможность легко добавлять новые виды анализа, было принято решение воспользоваться шаблоном проектирования «Посетитель» [4]. В роли посетителей выступают классы AnnotationValueTypeElementScanner и AnnotationValueTypeTreeScanner, а в роли узлов соответствующие потомки интерфейса Tree. За вычисление отношения вложенности типов отвечает реализация интерфейса SubtypingChecker, а за вычисление допустимости операции для уточняющего типа реализация интерфейса OperationPermissionChecker. За вывод сообщений об ошибках отвечает соответствующая реализация интерфейса Messager.

Учитывая разделение ответственности между интерфейсами, система обладает следующими возможностями расширения или изменения.

- Добавление новых видов анализа абстрактного синтаксического дерева программы.
- Добавление анализа для новых синтаксических узлов.
- Добавление новых видов ошибок и сообщений.
- Изменение подхода к ограничению или разрешению использования размеченных типов в операциях.

4.2. Статические проверки соответствия размеченных типов

Статические проверки, основанные на разработанном алгоритме анализа дерева разбора программы, проверяют корректное использование уточняющих типов в следующих случаях:

- при объявлении или вызове метода или конструктора;
- при объявлении локальной переменной или поля;
- при присваивании нового значения переменной или полю;
- при добавлении элемента в массив, тип которого размечен.

Пользователь может размечать аннотациями с уточняющим типом параметры в объявлении метода или конструктора, тип возвращаемого значения метода, а также типы в объявлениях локальных переменных и полей.

Статические проверки корректного использования размеченных типов в программе выполняются соответствующими методами классов AnnotationValueTypeElementScanner и AnnotationValueTypeTreeScanner, представленных на рис. 2. Методы классов анализируют узлы, реализующие интерфейсы Element и Tree соответственно. Абстрактное синтаксическое дерево программы, построенное компилятором Java, имеет несколько различных представлений, каждое из которых подходит для различных целей. В представлении, узлы которого имеют тип Element, листьями являются поля и методы классов и такое представление не содержит тел методов. А представление с узлами типа Tree является более низкоуровневым и позволяет анализировать выражения внутри тел методов. Причина, по которой для анализа используются именно эти представления, заключается в том, что процессор аннотаций получает доступ к классам исходного кода как к узлам типа Element. Затем, с помощью методов интерфейса ProcessingEnvironment, выделенного на рис. 2 синим цветом, для узлов Element извлекаются соответствующие им узлы Tree, которые анализируются.

Для того чтобы сократить число аннотаций необходимых для разметки исходного кода, локальные переменные поддерживают вывод уточняющего типа по инициализирующему выражению. Кроме того, для объявлений локальных переменных или полей представлена аннотация UnsafeCast, которая позволяет инициализировать переменную или поле значением несовместимого типа, генерируя при этом предупреждение. При необходимости предупреждение в месте использования можно отключить, передав соответствующий аргумент данной аннотации. Подобное не корректное с точки зрения уточняющих типов присваивание может быть полезно, например, при конвертации валюты.

4.3. Создание и использование иерархии подтипов

Проверка вложенности одного уточняющего типа в другой повсеместно используется в статических проверках корректного использования размеченных типов. За такие проверки отвечают методы интерфейса SubtypingChecker, представленного на рис. 2. Помимо этого, данный интерфейс содержит методы нахождения общего уточняющего типа, необходимые при анализе тернарного оператора. Для разметки типов в исходном коде используется аннотация Type, которой в качестве аргумента передается уточняющий тип. Для того чтобы использовать тип в качестве уточняющего, при его объявлении необходимо указывать аннотацию MetaType. Эта аннотация позволяет выделять типы, которые были объявлены с целью использования в качестве уточняющих типов. Пример объявления типа Encrypted вложенного в тип PossiblyUnencrypted представлен на рис. 3. Такой подход позволяет создавать иерархии подтипов, ограничиваясь привычным механизмом объявления производных классов в Java. Подобный подход к объявлению подтипов позволяет гарантировать соблюдение ограничений, накладываемых разработанным алгоритмом анализа. Единственность объемлющего типа гарантируется запретом множественного наследования в Java. А отсутствие циклов в иерархии подтипов гарантирует система типов Java.

```
@MetaType
class PossiblyUnencrypted {

}
@MetaType
class Encrypted extends PossiblyUnencrypted {

}

@Type(Encrypted.class)
private String password;
```

Рис. 3: Пример объявления и использования типов для разметки исходного кода

При объявлении уточняющих типов не обязательно указывать аннотацию MetaType, если какой-то из объемлющих уточняющих типов уже объявлен с этой аннотацией. Однако указание этой аннотации там, где это не является необходимым, увеличивает понимание исходного кода, так как явно указывает, какой цели служит объявленный тип.

4.4. Использование типов в операторах

Как уже было сказано, использование значений некоторых размеченных типов в арифметических, битовых или логических операциях может быть некорректным с точки зрения семантики этих типов. По умолчанию использование переменных размеченных типов в бинарных и унарных операторах запрещено. Для того чтобы разрешить использование размеченного типа в выбранном операторе, достаточно объявить уточняющий тип с соответствующей аннотацией, как показано на рис. 4.

За проверку применимости конкретного оператора для размеченного типа отвечает реализация интерфейса OperationPermissionChecker, представленного на рис. 2.

При объявлении уточняющих типов не обязательно указывать аннотацию соответствующей бинарной и унарной операции, если какой-то из

```java
@MetaType
@Plus @Minus
@Multiply @Divide
class Currency {
}

@MetaType
class Dollar extends Currency {
}

@MetaType
class Id {
}
```

```java
@Type(Dollar.class)
int evaluateSum(@Type(Dollar.class) int val1,
                @Type(Dollar.class) int val2) {
    return val1 + val2; // it's OK
}

@Type(Id.class)
void wrongUseId(@Type(Id.class) int id) {
    //...
    printId(id + id); // error!
    //...
}
```

Рис. 4: Пример объявления и использования типов с арифметическими операциями

объемлющих уточняющих типов уже объявлен с этой аннотацией. Такой подход позволяет сократить число аннотаций, которые необходимо указывать пользователю, а также гарантировать, что каждый объявленный вложенный тип может безопасно использоваться во всех операторах, в которых может использоваться объемлющий его тип.

4.5. Вывод сообщений об ошибках

Немаловажную роль в использовании программы играет система оповещения об ошибках, а информативные сообщения значительно ускоряют их поиск и исправление. За формирование и вывод сообщений с ошибками или предупреждениями при использовании размеченных типов отвечает реализация интерфейса Messager, изображенного на рис. 2. Сообщения содержат информацию о типах и операторах, в использовании которых возникла ошибка. Кроме того, при использовании IDE, например, IntelliJ IDEA нажатие на сообщение показывает место возникновения ошибки. Пример вывода сообщений с ошибками и предупреждениями демонстрируется на рис. 5.

```java
@Type(Euro.class)
public long moneyInEuros;

public void add(@Type(Euro.class) int sum) {
    this.moneyInEuros += sum;
}

public void debit(@Type(Euro.class) int sum) {
    this.moneyInEuros -= sum;
}

public void transferFrom(BankAccount bAcc) {
    @Type(Dollar.class)
    int sum = 1000;
    bAcc.debit(sum); // error; argument mismatch: 'Dollar'
                     // cannot be converted to 'Euro'

    this.add(sum);// error; argument mismatch: 'Dollar'
                  // cannot be converted to 'Euro'
}
```

```
examples/src/main/java/examples/BankAccount.java 2 errors
    argument mismatch: 'examples.Dollar' cannot be converted to 'examples.Euro'
    argument mismatch: 'examples.Dollar' cannot be converted to 'examples.Euro'
```

Рис. 5: Пример вывода сообщений об ошибках

5. Апробация реализованного инструмента

Для апробации системы необходимо было выбрать проект, разметить его код необходимыми аннотациями и запустить статический анализ. В качестве такого проекта был выбран XChange [11]. XChange — это библиотека Java с открытым исходным кодом, предоставляющая оптимизированный API для взаимодействия с более чем шестьюдесятью биржами криптовалюты, обеспечивающая согласованный интерфейс для торговли и доступа к рыночным данным. В этом проекте определенно множество классов, методы которых используют несколько однотипных параметров. Одним из таких классов является класс

[11]https://github.com/knowm/XChange

Balance, представляющий баланс пользователя в заданной валюте. Перед запуском статических проверок был размечен конструктор класса Balance, использующий семь параметров типа BigDecimal, а также 128 мест его использования. Для определения уточняющих типов потребовалось 70 дополнительных строк кода. Для разметки параметров в объявлении и аргументов в местах использования потребовалось около 900 аннотаций. В ходе статического анализа в 128 местах использования ошибок не было обнаружено.

Большое количество аннотаций, потребовавшихся в ходе проведения апробации, объясняется особенностью использования данного конструктора. Во всех 128 анализируемых местах использования конструктор вызывался с уникальными параметрами, поэтому потребовалось каждый раз размечать все семь переменных, использующихся в качестве аргументов. Отсутствие ошибок в ходе анализа объясняется развитием библиотеки XChange. Этот проект имеет большой круг пользователей, что несомненно оказывает влияние на обнаружение ошибок, и за многие годы существования проекта большинство из них наверняка были найдены и исправлены.

Заключение

В ходе данной работы были достигнуты следующие результаты.

1. Сделан возможных подходов к решению возникшей проблемы, а также существующей системы.

2. Разработан алгоритм анализа аннотаций в исходном коде программы.

3. На основе разработанного алгоритма, реализовано инструментальное средство, позволяющее

   (a) размечать типы уточняющими аннотациями;

   (b) статически проверять корректность использования размеченных типов;

   (c) создавать иерархию подтипов на размеченных типах;

   (d) ограничивать использование размеченных типов в выбранных бинарных и унарных операторах.

4. Проведена апробация реализованного инструментального средства.

Список литературы

[1] Buckley Alex. JSR 175: A Metadata Facility for the Java Programming Language. — 2004. — URL: https://jcp.org/en/jsr/detail?id=175 (дата обращения: 22.10.2019).

[2] The Checker Framework Manual: Custom pluggable types for Java. — 2019. — URL: https://checkerframework.org/manual/checker-framework-manual.pdf (дата обращения: 14.11.2019).

[3] Darcy Joseph D. JSR 269: Pluggable Annotation Processing API. — URL: https://jcp.org/en/jsr/detail?id=269 (дата обращения: 04.12.2019).

[4] E. Gamma R. Helm R. Johnson J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. — Addison-Wesley, 1994.

[5] Erni David, Kuhn Adrian. The Hacker's Guide to Javac // Technical report, Software Composition Group, University of Bern, Switzerland. — 2008. — http://scg.unibe.ch/archive/projects/Erni08b.pdf.

[6] Ernst Michael D. Ali Mahmood. Building and using pluggable type systems // Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering (FSE). — 2010.

[7] Goetz Brian. The pseudo-typedef antipattern: Extension is not type definition. — 2006. — URL: https://www.ibm.com/developerworks/java/library/j-jtp02216/ (дата обращения: 04.11.2019).

[8] Grigore Radu. Java generics are turing complete // In Symposium on Principles of Programming Languages (POPL). — 2017.

[9] J. Gosling B. Joy G. Steele G. Bracha A. Buckley D. Smith. The Java Language Specification. Java SE 11 Edition. — URL: https://docs.oracle.com/javase/specs/jls/se11/html/index.html (дата обращения: 22.10.2019).

[10] Jean-Marie Dautelle Werner Keil Otavio Santana. JSR 385: Units of Measurement API 2.0. — URL: https://www.jcp.org/en/jsr/detail?id=385 (дата обращения: 14.11.2019).

[11] Kennedy Andrew, Pierce Benjamin C. On Decidability of Nominal Subtyping with Variance // In Workshop on Foundations and Developments of Object-Oriented Languages (FOOL/WOOD). — 2007.

[12] Mark E. Royer Sudarshan S. Chawathe. Java unit annotations for units-of-measurement error prevention // 2018 IEEE 8th Annual Computing and Communication Workshop and Conference (CCWC). — 2018. — P. 816–822.

[13] Oracle. Java Development Kit Version 11 API Specification. — URL: https://docs.oracle.com/en/java/javase/14/docs/api/index.html (дата обращения: 22.10.2019).

[14] Pierce Benjamin C. Types and Programming Languages. — The MIT Press, 2002.

[15] Schildt Herbert. Java. The Complete Reference. Eleventh Edition. — Oracle Press, 2019.