Предметно-ориентированные аннотации для типов в Java

Автор: Сокольвяк Сергей Дмитриевич, 444 группа(16.Б10-мм)
Научный руководитель: доц. кафедры СП, к. ф.-м. н. К. Ю. Романовский
Научный консультант: ст. преп. кафедры СП, Я. А. Кириленко
Рецензент: программист ООО "ИнтеллиДжей Лабс", к. ф.-м. н. Д. А. Березун

Санкт-Петербургский государственный университет
Кафедра системного программирования

13 июня 2020г.

Введение

- Системы типов позволяют эффективно предотвращать ошибки при разработке ПО
- Некоторые ошибки системы типов не могут обнаружить, особенно при использования примитивных типов для обозначения сущностей из предметных областей
- Уточнение типов и статические проверки позволят избегать подобных ошибок
- Для того чтобы проверки не были чрезмерно жесткими, необходимо определять подтипы

Системы типов являются эффективным языковым средством, позволяющим избегать многих ошибок. Однако некоторые ошибки системы типов не обнаруживают, например, перечисление однотипных аргументов в неправильном порядке или использование сущностей предметных областей, выраженных примитивным типом, в недопустимых для них операциях. Уточнение типов и статические проверки помогут избежать подобных ошибок. Чтобы статические проверки не были слишком жесткими и не запрещали использовать значение одного типа вместо другого, если это безопасно, необходимо определять подтипы.

Цель и задачи

Цель: уменьшить количество ошибок, возникающих при использовании примитивных типов для обозначения сущностей из разных предметных областей в языке Java
Задачи:

- Рассмотреть существующие подходы к решению проблемы
- Разработать алгоритм анализа, позволяющий находить упомянутые ошибки в программах
- Реализовать инструментальное средство, позволяющее
  - статически проверять корректность использования размеченных типов
  - создавать иерархию подтипов на размеченных типах
  - ограничивать использование размеченных типов в выбранных операциях
- Провести апробацию реализованного инструментального средства

Цель данной работы — уменьшить количество ошибок при использовании примитивных типов для обозначения сущностей предметных областей в языке Java. Для этого были выделены задачи, которые перечислены на слайде.

Выделение подходов к решению проблемы

- Изменение архитектуры: объявление новых типов с помощью Java классов
- Смена языка программирования на Kotlin
- Использование библиотек единиц измерения
- Использование аннотаций для уточнения типов в исходном коде

Для достижения поставленной цели были выделены подходы, указанные на слайде. Прокомментируем каждый из них. Для уточнения типов можно оборачивать их в новые классы. Однако при использовании такого подхода необходимо упаковывать и распаковывать значения при использовании сторонних библиотек, а проверки на равенство и эквивалентность могут возвращать неправильные результаты. Кроме того, использование классов вместо примитивных типов может снизить производительность программы. При использовании более современных языков программирования, например, Scala или Kotlin, подобные ошибки, возможно, не возникали бы, однако не все компании готовы к переходу на другой язык программирования, ведь это сопровождается издержками. Кроме того, Java — это один из самых популярных языков, на котором пишут много программистов. Библиотеки единиц измерения позволяют уточнять примитивные типы, однако 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. Об использовании аннотаций поговорим на следующем слайде.

Использование аннотаций

- Аннотации позволяют встраивать в код дополнительную информацию
- С помощью аннотаций можно размечать типы там, где это необходимо
- Аннотации не оказывают влияния на работу программы
- Существуют инструментальные средства для анализа аннотаций во время компиляции программы
- С помощью статических проверок аннотаций можно избегать упомянутых ошибок

Аннотации — это языковое средство Java. Они позволяют встраивать в код дополнительную информацию, с помощью которой, например, можно уточнять типы. Аннотации не оказывают влияния на исполнение и производительность программы. Существуют инструментальные средства для анализа аннотаций, с помощью которых можно выполнять необходимые статические проверки. О таком средстве анализа поговорим на следующем слайде.

Annotation Processing Tool

- Инструментальное средство для анализа аннотаций в исходном коде программы
- Анализ происходит на этапе компиляции
- Необходимо определить процессор аннотаций для заданного набора аннотаций и реализовать в нем необходимую логику анализа

Annotation Processing Tool — это инструментальное средство для обработки аннотаций в исходном коде программы. Анализ аннотаций является отдельной фазой компиляции программы на языке Java. Для того чтобы использовать возможности Annotation Processing Tool для обработки аннотаций, необходимо определить собственный процессор аннотаций, реализующий логику анализа.

Похожая система. The Checker Framework

- Порядка 20 различных Checker-ов: Nullness, Regex, Initialization, Subtyping и др.
- Громоздкая система, содержит в себе много «лишнего»
- Нельзя ограничить использование типов в операторах
- Сложно добавить новую функциональность
- На момент начала данной работы поддерживал только JDK6

Checker Framework — это система, основанная на возможностях Annotation Processing Tool и позволяющая анализировать программы на Java. Состоит из 20-ти компонент, которые могут находить разнообразные ошибки, например, использование пустых ссылок, несоответствие строки шаблону. В частности, позволяет размечать код аннотациями для выполнения дополнительной проверки соответствия типов. Эта система позволяет частично решать поставленные задачи, однако имеет несколько значительных недостатков. Это громоздкая система, которую сложно интегрировать и расширять, а большая часть ее функциональных возможностей не относится к поставленной задаче. Не предоставляет возможности запрещать использование типов в операциях. На момент начала данной работы поддерживала только JDK6. На следующем слайде поговорим об алгоритме анализа исходного кода, на котором основано разработанное нами инструментальное средство.

Алгоритм анализа исходного кода программы

- Основан на применении *принципа безопасной подстановки*
- Алгоритм принимает на вход абстрактное синтаксическое дерево исходного кода программы
- Анализ заключается в последовательном определении размеченных типов узлов абстрактного синтаксического дерева
- Если в какой-то момент выполнения алгоритма невозможно вывести тип синтаксического терма, то выполнение останавливается и возвращается соответствующая ошибка компиляции

Алгоритм анализа основан на применении принципа безопасной подстановки, то есть проверки, что фактический тип выражения может использоваться вместо типа, который ожидается в контексте. На вход принимает абстрактное синтаксическое дерево программы. Анализ заключается в последовательном определении размеченных типов узлов дерева разбора. Если в какой-то момент выполнения алгоритма невозможно вывести тип синтаксического терма, то выполнение останавливается и возвращается соответствующая ошибка компиляции.

Правила вывода типа некоторых синтаксических термов

Типизация выражений с бинарным оператором:

$$\frac{bin\_op(left : T, right : S) \quad S <: T \quad bin\_op \text{ разрешен для } T}{bin\_op(left, right) : T}$$

Типизация вызовов методов и конструкторов:

$$\frac{func : T_1,\ T_2,\ \ldots \rightarrow R \qquad S_1 <: T_1,\ S_2 <: T_2,\ \ldots}{func\_invok(S_1,\ S_2,\ \ldots) :}$$

Типизация объявлений локальных переменных и полей:

$$\frac{identifier : T = init : S \qquad S <: T}{var\_decl : T}$$

Как уже было сказано, алгоритм анализа основан на выводе размеченных типов синтаксических термов. На слайде представлены правила вывода типа для некоторых часто встречающихся синтаксических термов, записанные в следующей нотации. Над чертой записаны предпосылки правила, под чертой — суждения. На следующем слайде поговорим об архитектуре разработанного инструментального средства.

Архитектура системы

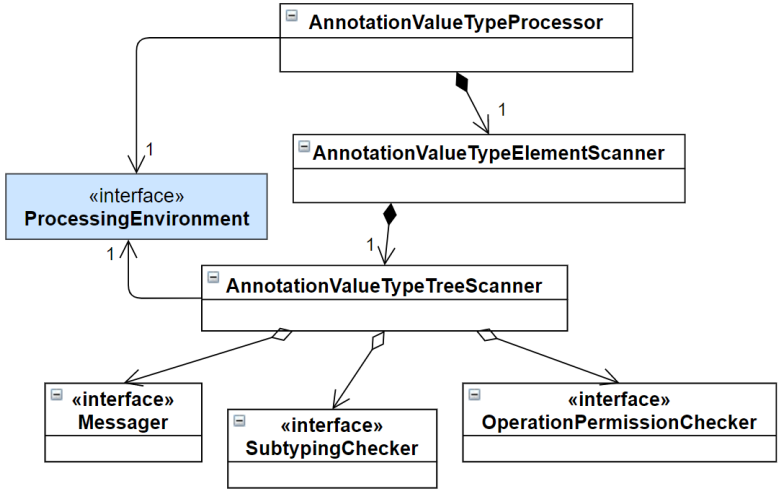

Рис.: Архитектура системы

Для реализации описанного алгоритма была разработана следующая архитектура системы. Класс AnnotationValueTypeProcessor является процессором аннотаций. Анализ дерева разбора реализуют классы AnnotationValueTypeTreeScanner и AnnotationValueTypeElementScanner. ProcessingEnvironment предоставляет доступ к абстрактному синтаксическому дереву анализируемой программы. За проверку отношения вложенности типов отвечает SubtypingChecker. Проверку допустимости операции для типа выполняет OperationPermissionChecker. Оповещение об ошибках выполняют методы Messager-а.

Статические проверки

Для обеспечения необходимых статических проверок пользователь может размечать исходный код в следующих местах:

- типы параметров методов и конструкторов
- тип возвращаемого значения метода
- тип объявленной локальной переменной или поля

На слайде указано, в каких местах пользователь может использовать аннотации с уточняющим типом для того, чтобы избегать описанных ошибок.

Определение и использование подтипов

- Для определения уточняющих типов используются Java классы
- Для определения вложенного типа используется наследование
- Отношение вложенности типов является *рефлексивным*: $S <: S$ и *транзитивным*: $\frac{S<:U \quad U<:T}{S<:T}$
- Задача определения вложенности типов разрешима, т.к.
  - в качестве уточняющих типов не используются параметризованные типы
  - иерархия подтипов *ациклична*

Для объявления уточняющих типов используются Java классы, а для построения иерархии вложенных типов наследование этих классов. Такой подход позволяет гарантировать ацикличность иерархии. Циклы возникают, когда последовательное применение правила транзитивности для произвольного типа S позволяет получить суждение о том, что S является подтипом S. Исходя из результатов исследований, описанных в статье “On Decidability of Nominal Subtyping with Variance“ , ацикличность иерархии подтипов и отсутствие параметризованных типов позволяют гарантировать разрешимость задачи определения подтипов.

Использование размеченных типов в операторах

- По умолчанию использование размеченных типов в бинарных и унарных операторах запрещено
- Указание аннотаций при объявлении уточняющего типа разрешает его использование в соответствующих операциях
- При определении вложенного типа неявно указываются аннотации его объемлющего типа

Если тип разрешено использовать в какой-либо операции, то и любой его подтип тоже может использоваться в этой операции и это нельзя запретить. В противном случае нарушается принцип безопасной подстановки. Исходя из этого, было принято решение по умолчанию запретить использование размеченных типов в операциях. Для того чтобы разрешить использование в конкретной операции, необходимо указать соответствующую аннотацию при объявлении. Чтобы гарантировать, что вложенный тип может использоваться во всех операциях, где может использоваться объемлющий тип, а также сократить число аннотаций в коде, эти аннотации наследуются и их необязательно указывать в подтипах.

Вывод сообщений об ошибках

```java
/*
Представляет объект, который может быть не зашифрован
 */
@MetaType
@Plus
public class PossiblyUnencrypted {
}
/*
Представляет объект, который должен быть зашифрован
 */
public class Encrypted extends PossiblyUnencrypted {
}
// только для зашифрованных данных
public void sendOverInternet(@Type(Encrypted.class) String msg) {
    // ...
}
void sendText() {
    // ...
    @Type(Encrypted.class) String encryptedText = encrypt(plaintext);
    sendOverInternet(encryptedText); // здесь нет ошибки
    // ...
}
void sendPassword() {
    String password = getUserPassword();
    sendOverInternet(password); // здесь ошибка
}
```

```
src/main/java/test/MainExample.java 1 error
argument mismatch: 'raw type' cannot be converted to 'test.qual.Encrypted'
```

Рис.: Пример вывода сообщений об ошибках

Пример вывода сообщений об ошибках представлен на слайде. Сообщения об ошибках содержат информацию, необходимую для понимания сути ошибки. Они указывают на место в коде, содержат названия размеченных типов и причину, по которой использование указанных типов в данном контексте некорректно. Подобная информация может быть полезна при отладке программы.

Апробация реализованной системы

1. В качестве тестового проекта была выбрана open-source библиотека XChange
2. Разметили параметры конструктора класса Balance, представляющего баланс в валюте, а также аргументы в местах вызова конструктора. Конструктор содержит семь BigDecimal значений
3. Конструктор вызывается в коде проекта 128 раз
4. Для определения дополнительных типов потребовалось порядка 70 строчек кода
5. Апробация не выявила ошибок в вызовах конструктора

Для апробации был выбран open-source проект XChange. Это библиотека для взаимодействия с более шестьюдесятью биржами криптовалюты и торговли на них. Результаты апробации указаны на слайде. Отсутствие ошибок в ходе анализа можно объяснить, например, развитием библиотеки XChange. Этот проект имеет большой круг пользователей и за многие годы существования проекта большинство из ошибок наверняка были исправлены.

Пример размеченного кода XChange

```java
public static AccountInfo adaptAccountInfo(CoingiBalances coingiBalances, String userName) {
  List<Balance> balances = new ArrayList<>();
  for (CoingiBalance coingiBalance : coingiBalances.getList()) {
    @Type(Total.class)
    BigDecimal total =
        coingiBalance
            .getAvailable()
            .add(coingiBalance.getBlocked())
            .add(coingiBalance.getWithdrawing())
            .add(coingiBalance.getDeposited());
    Balance xchangeBalance =
        new Balance(
            Currency.getInstance(coingiBalance.getCurrency().getName().toUpperCase()),
            total, // total = available + frozen - borrowed + loaned + withdrawing + depositing
            coingiBalance.getAvailable(), // available
            coingiBalance.getBlocked(),
            BigDecimal.ZERO, // borrowed is always 0
            BigDecimal.ZERO, // loaned is always 0
            coingiBalance.getWithdrawing(),
            coingiBalance.getDeposited());
```

Рис.: Пример размеченного кода

Несмотря на то, что апробация не выявила ошибок в использовании конструктора, инструментальное средство позволило бы избежать их, например, при неправильном порядке перечисления размеченных аргументов. На слайде представлен пример использования конструктора Balance из проекта XChange, в котором потенциально могла бы возникнуть такая ошибка.

Результаты

- Сделан обзор возможных подходов к решению возникшей проблемы
- Разработан алгоритм анализа исходного кода программы
- На основе придуманного алгоритма реализовано инструментальное средство
  1. позволяющее размечать типы в исходном коде Java
  2. проверяющее корректность использования размеченных типов
  3. позволяющее создавать иерархию подтипов на размеченных типах
  4. выводящее сообщения об ошибках
- Проведена апробация реализованного инструментального средства

На слайде представлены результаты, достигнутые в ходе данной работы.