Оглавление

1. Введение

1.1 Статический анализ кода

С течением времени сложность программных систем возрастает. Как следствие, их тестирование занимает все больше времени. При многопоточном программировании особенно часто проявляются ошибки, сложные для обнаружения, воспроизведения и анализа. Вся логика работы программы уже не может быть легко понята и осмыслена одним программистом. В связи с дороговизной ручного тестирования и большой сложностью программыных систем активно развиваются различные виды автоматической проверки программ. Их можно условно разделить на два вида:

- автоматическая проверка корректности программы во время ее работы или работы ее отдельных частей,
- проверка программы на корректность без ее запуска.

К первой группе можно отнести всевозможные виды тестирования.

Определение 1.1.1. *Статический анализ кода – это анализ программного обеспечения, производимый без реального выполнения исследуемых программ.*

Статический анализ позволяет еще до запуска программы выявить многие виды ошибок, большинство из которых сложно искать и воспроизводить непосредственно во время работы приложения. В связи с этим актив-

но развиваются различные инструменты, позволяющие статически доказывать отсутсвие в программах ошибок тех или иных видов.

1.2 Неизменяемость в контексте объектно-ориентированного языка

В различных контекстах понятие неизменяемости может пониматься по-разному. В данной работе рассмотрено несколько видов неизменяемости:

Определение 1.2.1. *Неизменяемый класс – класс, все экземпляры которого являются неизменяемыми.*

Определение 1.2.2. *Неизменяемая ссылка – ссылка, которая не может быть использована для изменения объекта, на который она указывает (при этом объект может быть изменен через другую ссылку).*

Если какая-либо система статического анализа позволяет выражать данное свойство объекта, будем говорить, что в данной системе есть поддержка *ссылочной неизменяемости.*

Определение 1.2.3. *Неизменяемый объект – объект, о котором известно, что не существует ни одной ссылки, через которую его можно изменить.*

Если какая-либо система статического анализа позволяет выражать данное свойство объекта, будем говорить, что в данной системе есть поддержка *объектной неизменяемости.*

Определение 1.2.4. *Глубокая (транзитивная) неизменяемость характеризует следующее свойство: если объект неизменяем, то неизменяемы также и все объекты, доступные из него по ссылкам.*

1.3 Обзор существующих решений

Рассмотрим, как проблема контроля изменяемости решается в различных объектно-ориентированных языках программирования.

1.3.1 C++

В языке C++ для выражения неизменяемости используется ключевое слово const. В общем случае можно сказать, что если какое-то значение неизменяемо, то в ту часть памяти компьютера, где оно хранится, не может быть произведена запись. В случае с нессылочными типами данных, если какая-либо переменная объявлена как const, то ее значение не может быть изменено после инициализации. Это означает, что в C++ есть объектная неизменяемость.

Листинг 1.1: Константая переменная

```
1  struct S
2  {
3      int val;
4  };
5
6
7  const S const_s;
8  const_s.val = 42;       // Error: const_s was declared as const
9  int i  = const_s.val;   // OK: field val is accessed for
10                         // reading, not for writing
11
12 const S non_const_s;
13 non_const_s.val = 42;   // OK: non_const_s was not
14                         // declared as const
```

Для указателей и ссылкок значение модификатора const более сложное. Константным может быть сам указатель, значение, на которое он указывает или оба. Если какая-либо переменная объявлена как константый

указатель, то ее значение не может быть изменено после инициализации. Если переменная объявлена как указатель на константный объект, то ее значение может быть изменено, но ее нельзя использовать для изменения объекта, на который она указывает. Таким образом, в C++ есть ссылочная неизменяемость. Нужно заметить, что не существует никакого способа сказать, что некий указатель указывает на неизменяемый объект. Все то же самое касается ссылок.

Листинг 1.2: Константный указатель

```
1  struct S
2  {
3      int val;
4  };
5
6  void Foo( S * ptr,
7            S const * ptrToConst,
8            S * const constPtr,
9            S const * const constPtrToConst )
10 {
11     ptr->val = 0;     // OK: modifies the "pointee" data
12     ptr  = NULL;      // OK: modifies the pointer
13
14     ptrToConst->val = 0; // Error: cannot modify
15                          // the "pointee" data
16     ptrToConst  = NULL;  // OK: modifies the pointer
17
18     constPtr->val = 0; // OK: modifies the "pointee" data
19     constPtr  = NULL;  // Error: cannot modify the pointer
20
21     constPtrToConst->val = 0; // Error: cannot modify
22                               // the "pointee" data
23     constPtrToConst  = NULL;  // Error: cannot modify
24                               // the pointer
25 }
```

Методы, которые не изменяют значение объекта, на котором вызываются, могут быть помечены ключевым словом const. Тот факт, что метод действительно не изменяет объект, на котором вызывается, проверяется статически. Методы, помеченные как const могут быть вызваны как у константных ссылок, так и у неконстантных ссылках. Методы, не помеченные как const, могут быть вызваны только на неконстантных ссылках.

В даном подходе есть несколько недостатков. Первый из них связан с хранением в объекте указателей на другие объекты. Если некий объект является константным, то указатели, хранящиеся в нем в качетсве полей, будут константными, но при этом они могут быть использованы для изменения объекта, на который ссылаются. Рассмотрим пример:

Листинг 1.3: Пример изменения значения по указателю в константном методе

```
1  struct S
2  {
3      int val;
4      int *ptr;
5  };
6
7  void Foo(const S & s)
8  {
9      int i  = 42;
10     s.val  = i;  // Error: s is const, so val is a const int
11     s.ptr  = &i; // Error: s is const, so val is a const int
12     *s.ptr = i;  // OK: the data pointed to by ptr
13                  // is always mutable
14 }
```

Несмотря на то, что s передается в метод Foo() как константый (что также делает константными всех его членов), объект, доступный через s.ptr можно изменять. Таким образом, в C++ нет поддержки глубокой неизменяемости.

Также в C++ невозможно вернуть ссылку, чья изменяемость зависит от изменяемости this. Поэтому, например, во всех колелкциях STL содержатся по две перегруженные версии iterator и operator[], которые, фактически, делают одно и то же, отличаясь только константностью и, как следствие, константностью возвращаемого значения.

1.3.2 C#

В C# ключевое слово readonly, примененное к полям имеет слелующий смысл: присвоение значения полю, которое было объявлено с модификатором readonly может произойти либо по месту его объявления, либо в конструкторе, если это нестатическое поле (для статического поля - в статическом конструкторе).

Листинг 1.4: Ключевое слово readonly в C#

```
1  using System;
2  public class ReadOnlyTest
3  {
4      class MyClass {
5          public int x;
6          // Initialize a readonly field
7          public readonly int y = 25;
8          public readonly int z;
9          public MyClass() {
10             // Initialize a readonly instance field
11             z = 24;
12         }
13
14         public MyClass(int p1, int p2, int p3) {
15             x = p1;
16             y = p2;    // OK: readonly field can
17                        // be reassigned in constructor
18             z = p3;
19         }
```

```
20      }
21
22      public static void Main() {
23          MyClass p2 = new MyClass();
24          p2.x = 55;   // OK: field x is not readonly
25          p2.y = 33;   // Error: field y can't be
26                       // reassigned, as it is readonly
27      }
28  }
```

В C# также есть ключевое слово const, которое обозначает, что значение перменной может быть присвоено только в момент ее объявления. То есть, поля объекта, объявленные как readonly, могут иметь различные значения в зависимости от того, какой конструктор и с какими параметрами был вызван. Поле, объявленное как const всегда будет иметь одно и то же значение.

Листинг 1.5: Ключевое слово const в C#

```
1   using System;
2   public class ReadOnlyTest
3   {
4      class MyClass
5      {
6         public int x;
7         public const int y = 25; // Initialize a const field
8
9         public MyClass() {
10           z = 24;   // Initialize a readonly instance field
11        }
12
13        public MyClass(int p1, int p2) {
14           x = p1;
15           y = p2;   // Error: const field can not be
16                     // reassigned in constructor
17        }
```

```
18      }
19
20      public static void Main() {
21          MyClass p2 = new MyClass();
22          p2.x = 55;   // OK: field x is not readonly
23          p2.y = 33;   // Error: field y can't be reassigned,
24                       // as it is const
25
26      }
27  }
```

1.3.3 Java

В языке Java есть ключевое слово final, обозначающее, что значение соответствующего поля или переменной не может быть переприсвоено. Если все поля некоторого объекта объявлены как final, то можно говорить о том, что данный объект неизменяем. Действительно, после завершения конструктора в final поле всегда может находиться один и тот же объект, но сам этот объект может быть изменен. Таким образом, в Java нет поддержки глубокой неизменяемости.

Листинг 1.6: Ключевое слово final

```
1  public class MyClass {
2      public final int[] values;
3
4      public MyClass() {
5          values = new int[10];
6      }
7  }
8
9  MyClass mc = new MyClass();
10 mc.values = new int[100]; // Error: field values
11                           // was declared final
12 mc.values[2] = 4; // OK: values is declared final,
```

```
13                      // but we can still change object,
14                      // referenced by this field.
```

Показательным является следующий пример: пусть есть некий класс, который содержит в себе ссылку на список объектов. Разработчик интерфейса этого класса хочет разрешить клиенту получать хранимый список, но не хочет, чтобы клиент мог модифицировать данный список. На Java код такого класса будет скорее всего выглядеть следующим образом:

Листинг 1.7: Неизменяемый список

```
1 public class ListContainer {
2     private final List<String> values = new
         ArrayList<String>();
3
4     public List<String> getValues() {
5         return Collections.unmodifiableList(values);
6     }
7 }
```

В данном случае, Collections.unmodifiableList(values) вернет обертку над исходным списком, у которой все изменяющие список методы переопределены так, что они бросают UnsupportedOperationException. Основным недостатком данного подхода является то, что ошибка будет обнаружена только во время выполнения программы. Ее локализация и исправление потребуют горадо больше усилий, чем если бы данная ошибка была выявлена на этапе компиляции.

Листинг 1.8: Использование неизменяемого списка

```
1 ListContainer container = new ListContainer();
2 List<String> cVals = container.getValues();
3 int size = cVals.size(); // OK: getting size is permitted for
4                          // immutable list
5 cVals.add("Hello!");     // Error: this code will be
6                          // successfuly
7                          // compiled, but will cause
```

```
8                               // UnsupportedOperationException
9                               // on runtime
```

Таким образом, в стандратной библиотеке Java неизменяемые коллеции реализованы просто как обертки над стандартыми интерфейсами, у которых переопределены изменяющие объект методы. Так как при вызове Collections.unmodifiableList() копирования элементов не происходит, то все изменения, сделанные в исходдной коллекции, будут "видны"в containerValues. Таким образом, результат работы метода ListContainer.getValues() является в некотором смысле неизменяемой ссылкой – через эту ссылку нельзя менять объект, но существуют другие ссылки на данный объект, через которые его можно менять.

Можно ли каким-либо образом избежать возможной ошибки времени исполнения, при этом не позволяя пользователю добавлять элементы в список values, хранящийся в ListContainer? Можно переписать класс ListContainer следующим образом:

Листинг 1.9: Неизменяемый список

```
1 public class ListContainer {
2     private final List<String> values = new
         ArrayList<String>();
3
4     public List<String> getValues() {
5         return new ArrayList<String>(values);
6     }
7 }
```

В этом случае, будет создана независимая копия списка values, которая и будет возвращена пользователю. Любые изменения, производимые с этой копией, не затронут исходный список.

Альтернативный подход можно наблюдать на примере библиотеки gs-collections [1]. В ней есть две отдельные иерархии коллеций – для изменяемых

[1] https://github.com/goldmansachs/gs-collections

коллекций и для неизменяемых. Таким образом, попытка вызвать изменяющий коллекцию метод на неизменяемой коллекции приведет к ошибке компиляции.

Листинг 1.10: Неизменяемый список

```
1 public class ListContainer{
2     private final MutableList<String> values = new
        FastList<String>();
3
4     public ImmutableList<String> getValues() {
5         return values.toImmutable();
6     }
7 }
```

С одной стороны, этот подход позволяет избежать ошибок, связанных с неправомерным изменением объектов во время выполнения программы, но с дургой его реализация требует написания гораздо большего количества кода. В такоем подходе есть еще одна проблема: структура интерфейсов и их реализации полностью определяются создателем библиотеки и ее пользователь имеет гораздо меньше свободы в ее использовании.

Часто в документации к Java-коду можно встретить информацию о том, что фактически объект, возвращаемый каким-либо методом является неизменяемым. Или, например, в документации к интрфейсу Map сказано, что "необходима большая осторожность при испрользовании изменяемых объектов в качестве ключей". Наличие подобной информации в документации показывает, что выразительных средств языка не хватает для выражения утверждений об изменяемости объектов и что, если бы подобные средства существовали, они могли бы быть востребованы.

1.3.4 Javari

Javari [1] – это расширение языка Java, которое добавляет в Java ссылочную неизменяемость, комбинируя статические и динамические провер-

ки неизменяемости. Авторы вводят следующее определение:

Определение 1.3.1. *Абстрактное состояние объекта – это состояние самого объекта и все достижимые из него по ссылкам состояния.*

Javari предоставляет гарантии относительно всего транзитивно достижимого состояния объекта – то есть, состояния самого объекта и состояний всех объектов, доступных из него по нестатическим ссылкам. При этом некоторые части класса могут быть исключены из его абстракного состояния.

Javari добавляет к Java пять дополнительных ключевых слов assignable, readonly, mutable и romaby. Рассмотрим их использование на примерах.

Пусть, например, переменная rodate имеет тип readonly Date. Тогда rodate не может быть быть использована только для тех операций, которые не меняют объект, на который ссылкается rodate:

Листинг 1.11: Неизменяемая ссылка

```
1 readonly Date rodate = ...; // readonly reference to a Date
     object
2 rodate.getMonth();    // OK
3 rodate.setYear(2005); // Error
4
5 /*mutable*/ Date date = new Date(); // mutable Date
6 rodate.getMonth();    // OK
7 rodate.setYear(2005); // Error
```

Пусть в Java существует некий ссылочный типа T. Тогда readonly T в Javari является супертипом T. Изменяемая ссылка может быть использована везде, где ожидатся неизменяемая ссылка. Это связано с тем, что неизменяемая ссылка только лишь запрещает менять объект, на который она ссылается, при этом ничего относительно этого объекта не гарантируя.

На данном рисунке представлена иерархия типов в Javari. Система типов гарантирует, что изменяющие объект методы не могут быть вызваны на неизменяемых ссылках.

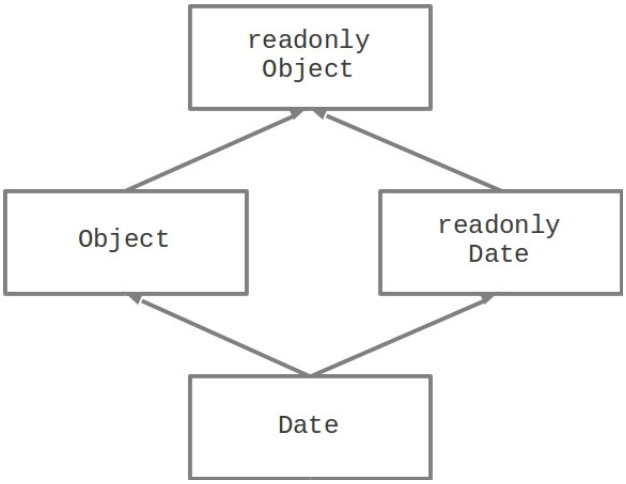

Рис. 1.1: Фрагмент иерархии классов в Javari

Ключевое слово readonly может быть спользовано при декларации любой переменной, поля, параметра или возвращаемого значения метода. Его также можно применять к неявному параметру this:

Листинг 1.12: readonly метод

```
1 public char charAt(int index) readonly { ... }
```

В контексте этого метода this будет неизменяемым.

Модификаторы изменяемости, введенные в Javari не меняют поведения программы во время исполнения. Такой подход обеспечивает обратную совместимость файлов, сгенерированных Javari, с файлами, сгенерированными обычным javac. Одним из последствий такого подходя является то, что два перегруженных метода не могут отличаться только изменяемостью их параметров. Например, такие два метода не могут перегружать друг друга:

Листинг 1.13: Перегрузка методов

```
1 void foo(/*mutable*/ Date d) { ... }
2 void foo(readonly Date d) { ... }
```

Это аналогично тому, что в Java два перегуженных метода не могут отличаться только типовыми параметрами.

Javari также позволяет исключать некоторые поля из абстрактного состояния объекта. По умолчанию все поля являются частью абстрактного

состояния объекта и, соответсвенно, не могу быть изменены через неизменяемую ссылку. Если поле объявлено как assignable, то его значение всегда может быть переприсвоено (даже через read-only ссылку). Ключевое слово mutable означает, что поле может быть изменено даже через неизменяемую ссылку. Это может быть полезно для кэширования данных или, например, для реализации логирования, как в следующем примере:

Листинг 1.14: assignable и mutable поля

```
1  class Foo {
2      assignable int hc;
3      final mutable List<String> log = new ArrayList<String>;
4
5      int hashCode() readonly {
6          log.add("hashCode invoked");
7          if (hc == 0) {
8              hc = ... ;
9          }
10         return hc;
11     }
12 }
```

Javari также позволяет добавлять модификаторы изменяемости к типовым параметрам:

Листинг 1.15: Модификаторы изменяемости в типовых параметрах

```
1 /*mutable*/ List</*mutable*/ Date> ld1; // add/remove and
2                                          // mutate elements
3 /*mutable*/ List<readonly Date> ld2; // add/remove
4 readonly List</*mutable*/ Date> ld3; // mutate elements
5 readonly List<readonly Date> ld4; // (neither)
```

Можно представить себе ситуацию, когда программисту захочется управлять изменяемостью типового параметра: например, написать mutable X, где X – типовый параметр:

Листинг 1.16: Модификаторы изменяемости и типовые параметры

```
1 class Container<X> {
2     void foo() {
3         mutable X x = ...;
4     }
5 }
```

Javari запрещает такие типы потому что это не сочетается с подходом к типовым параметрам, принятым в Java, и это может привести к превращении неизменяемой ссылки в изменяемую. Но в Javari, как и в Java, автор класса с типовым параметром может наложить на этот параметр границы. В примере ниже параметр X модет быть readonly Date, mutable Date или каким-либо из их наследников, в то время как Y может быть только mutable Date или его наследником.

Листинг 1.17: Объявление класса с типовыми параметрами

```
1 class Foo<X extends readonly Date, Y extends mutable Date> {
    ... }
```

Также Javari позволяет абстрагироваться от изменяемости типового параметра. Это можно сделать с помощью конструкции ? readonly C, где C – какой-то тип. Так, List<? readonly Date> является суперклассом для List<readonly Date> и List<mutable Date>.

Наконец, рассмотрим назначение ключевого слова romaybe. Пусть есть класс DateCell, который хранит в себе значение типа Date. Необходимо определить метод getValue, который будет возвращать это значение. Какого модификатор изменяемости должен стоять на возвращаемом значении? Если метод getValue вызывается на изменяемом объекте, то и его результат должен быть изменяемым. Если же он вызван на неизменяемом объекте, и результат его выполнения должен быть неизменяемым. Для решения этой проблемы в Javari было введено еще одно ключевое слово - romaybe. Так будет выглядеть класс DateCell с использованием этого ключевого слова:

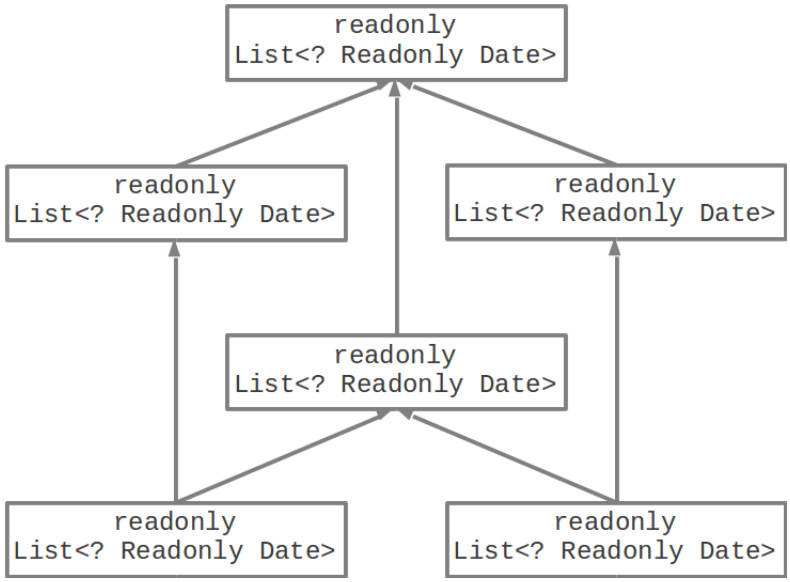

Рис. 1.2: Фрагмент иерархии классов в Javari

Листинг 1.18: Ключевое слово romaybe

```
1 class DateCell {
2     Date value;
3
4     romaybe Date getValue() romaybe {
5         return value;
6     }
7 }
```

В данной ситуации для системы типов существует два метода getValue: в первом все ключевые слова romaybe будут заменены на readonly, а во втором просто опущены.

Javari предоставляет инструмент под названием Javarifier, позволяющий добавить модификаторы изменяемости к уже существующему коду. На входе он принимает класс-файлы. В начале работы алгоритма некоторые поля помечаются как assignable или mutable (например, на основании того, что они меняются в методе hashCode). Данный алгоритм генерирует и решает систму утверждений для анализируемой программы. Используются два типа утверждений:

- некая ссылка является изменяемой: "x is mutable

- некая ссылка являтся измеянемой, если другая ссылка является изменяемой: "if y is mutable then x is mutable".

После составления системы утверждений алгоритм рашает ее.

Минусом данного подхода является то, что в общем случае количество уравнений а данной системе $O(n^2)$, где n - количесвто ссылок в анализируемом коде.

1.3.5 Immutability Generic Java

Immutablility Generic Java (IGJ) [2] – это расширение языка Java, которое позволяет выражать утверждения о неизменяемости объектов без внесения изменений в синтаксис Java, для этого IGJ использует типовые параеметры и аннотации. В IGJ каждый класс имеет дополнительный типовый параметр, который может иметь значения Immutable, Mutable или ReadOnly. IGJ поддерживает как объектную так и ссылочную неизменяемость. IGJ также разрешает ковариантные изменения типовых параметров в безопасной форме, например, неизменяемый список целых чисел является потомком неизменяемого списка чисел.

Рассмотрим применение данного расширения на примере:

Листинг 1.19: Пример использования IGJ

```
1  class Edge<I extends ReadOnly> {
2      private long id;
3
4      @AssignsField Edge(long id) {
5          this.setId(id);
6      }
7
8      @AssignsField synchronized void setId(long id) {
9          this.id = id;
10     }
11
```

```
12      @ReadOnly syncronized long getId() {
13          return id;
14      }
15
16      @Immutable long getIdImmutable() {
17          return id;
18      }
19
20      @ReadOnly Edge<I> copy() {
21          return new Edge<I>(id);
22      }
23
24      static void print(Edge<ReadOnly> e) {...}
25  }
26
27  class Graph<I extends ReadOnly> {
28      List<I, Edge<I>> edges;
29
30      @AssignsField Graph(List<I, Edge<I>> edges) {
31          this.edges = edges;
32      }
33
34      @Mutable void addEdge(Egde<Mutable> e) {
35          this.edges.add(e);
36      }
37
38      static <X extends ReadOnly> Edge<X>
39          findEdge(Graph<X> g, long id) {...}
40  }
```

В примере 1.19 представлены два IGJ класса: Edge и Graph. В строчках 1 и 27 определен параметр неизменяемости I. Если в декларации класса отсутсвует дирректива extends, считается, что класс наследует Object<I>. Присваивание this.id = id разрешено, так как метод setId помечен как Mutbale. Метод print, например, принимает любой объект типа Egde, вне

зависисмости от его изменяемости. Одним из плюсов IGJ является то, что тот факт, что поля некотго объекта имеют тот же параметр изменяемости, что и сам объект, легко выражается при помощи типовых параметров, как, например это сделано в строке 28. Например, в C++ поля, не обозначенные как const или mutable имеют модификатор неизменяемости такой же, как и у объекта, их содержащего, но при этом отсутсвует возможность указать, что какая-либо локальная переменная, параметр метода или возвращаемое значение имеют такую же неизменяемость, как и объект, на котором данный метод вызван, IGJ же предоставляет такую возможность.

Аннотация @AssignsField решает проблему с изменяемостью this в конструкторе. В конструкторе неизменяемого объекта нельзя счиать this Mutable, так как в этом случае было бы возможно, например, присвоить this в какую-либо глобальную @Mutable переменную и изменять объект уже после завершения его конструктора. Из метода, проаннотированного как AssignsField ссылка на this может быть присвоена только как ReadOnly ссылка.

Подход, описанный в данной работе, несомненно, позволяет выражать различные утверждения о неизменяемости объектов. Но у него есть несколько минусов:

- Получающийся код часто выглядит достаточно громоздко.
- Фаза конструирования объекта заканчивается тогда, когда заканчивает работу его конструктор. Это не позволяет достаточно легко создавать неизменяемые циклические структуры данных. Существующее расширение OIGJ [4] решает проблему с конструированием объектов, но делает это путем введения понятия владения объектом и, как следствие, добавлением еще одного типового параметра к каждому типу.
- Для того, чтобы эффективно использовать уже существующий код из классов, написанных с использованием IGJ необходимо вручную

добавить в него дополнительные типовые параметры, отвечающие за изменяемость объектов.

1.3.6 Язык D

Подход, похожий на тот, что был описан в IGJ используется в объекно-ориентированном языке D [2]. В D концепции объектной и ссылочной неизменяемости поддерживаются на уровне языка.

Во второй версии этого языка существует два ключевых слова для выражения неизменяемости: const и immutable. Ключевое слово immutable означает, что не существует ссылки, через которую данные могут быть изменены, const обозначает, что по данной ссылке данные менять нельзя, но может существовать ссылка, через которую данные могут быть изменены.

Листинг 1.20: const vs immutable

```
1  int[] foo = new int[5];      // foo is mutable.
2  const int[] bar = foo;       // bar is a const view of
3                               // mutable data.
4  immutable int[] baz = foo;   // Error:  all views of immutable
5                               // data must be immutable.
6
7  // No mutable reference to nums may be created.
8  immutable int[] nums = new immutable(int)[5];
9  // Immutable is implicitly convertible to const.
10 const int[] constNums = nums;
11 // Error:  Cannot create a mutable view of immutable data.
12 int[] mutableNums = nums;
```

В отличае от const в C++, const и immutable в D обеспечивают полноценную глубокую неизменяемость, то есть, любые данные, доступные через const или immutbale объект, также константны или неизменяемы, соответсвенно.

[2] http://dlang.org/

Листинг 1.21: const vs immutable

```
1 class Foo {
2     Foo next;
3     int num;
4 }
5
6 immutable Foo foo = new immutable(Foo);
7 foo.next.num = 5;  // Error:  foo.next is of type
8                    // immutable(Foo). foo.next.num
9                    // is of type immutable(int).
```

1.3.7 Безопасное параллельное выполнение при помощи контроля за уникальностью и ссылочной неизменяемостью

В работе Uniqueness and Reference Immutability for Safe Parallelism [3] представлено расширение для языка C#. Основной задачей этого расширения является ограничение изменений областей памяти при параллельном программировании. Это достигается компбинацией модификаторов изменяемости и уникальности. Система типов поддерживает полиморфизм относительно этих модификаторов, а также простое создание циклов неизменяемых объектов.

В рамках данной работы у каждой сслки может быть один из следующих модификаторов:

- writable – ссылка, позволябщая изменять объект, на который она ссылается;
- readable – неизменяемая ссылка, которая не позволяет изменять объект, на который она ссылается;
- immutbale – ссылка на неизменяемый объект;

- isolated – уникальная ссылка на кластер объектов (то есть, все пути в этот подраф объектов извне будут идти через эту ссылку);

Введение такого понятия, как isolated ссылка, повзоляет решить проблему с созданием неизменяемых циклических структур данных, так как если некая ссылка является едиснвтенной ссылкой на некий подграф объектов, то изменяемость этого подграфа может быть едитновременно безопасно изменена, так как не существует других ссылок на данный подграф.

В предложенном в данной работе языке отсутсвуют глобальные изменяемые переменные и поля, исключаемые из состояния объекта, что позволяет, напрмиер, говорить о том, что через readable ссылку не может быть достигнута никакая writable ссылка. Подобные ограничения полезны при анализе многопоточного поведения программы, но для статического контроля за изменяемостью они представляются слишком сильными. Например, введение подобный ограничений для Java означало бы отказ от не-final статических полей, а так же от хранения в статических полях объектов, чье стостояние может быть изменено. Также в данной работе никак не рассмотрен вопрос о том, каким образом при наличии всех этих ограниченйи могу бы быть использован уже существующий код.

1.4 Постановка задачи

Целью данной работы была разработка системы аннотаций, позволяющей контролировать изменяемость объектов на этапе компиляции для языка Java.

К данной системе были предъявлены следующие требования:

- Должна быть поддержана как объектная, так и ссылочная неизменяемость.

- Необходима возможность исключать некоторые поля из абстрактного

состояния объекта.

- Возможность создавать неизменяемые циклические структуры объектов.
- Возможность автоматически проаннотировать уже существующий код.

В рамках данной работы решались следующие задачи:

- Разработка системы аннотаций, позволяющей выражать неизменяемость объектов.
- Разработка алгоритма вывода аннотаций для существующего кода.

2. Статический контроль за изменяемостью объектов

В данном разделе представлена разработанная система аннотаций, а также описан алгоритм, позволяющий вывести эти аннотации для уже существующего библиотечного кода.

2.1 Подход к технической реализации

Предположим, нужно добавить некоторую новую функциональность в язык программирования. Есть два принципиально разных способа это сделать:

- использовать существующие средства языка;
- изменять синтаксис языка (например, добавить новые ключевые слова);

У обоих этих подходов есть как положительные, так и отрицательные стороны. Изменение синтаксиса языка приводит к невозможности использования многих существующих инструментов для разработки с использованием этого языка, таких как компиляторы, среды разработки, различные анализаторы кода. Но с другой стороны этот подход позволяет добавлять в язык развитую систему выразительных средств. Использование же существующих средств языка ограничивает свободу введения новых концепций,

но этот подход обычно гораздо проще в реализации и не влияет на используемые инструменты.

В случае Java есть несколько способов добавить поддержку неизменяемости объектов в язык. В работе IGJ это сделано с помощью добавления дополнительного типового параметра ко всем классам. Но это выглядит очень громоздко и трудно читаемо. Другой варинат – использование аннотаций.

Аннотация в Java – это вид метаданных, которые могут быть добавлены в исходный код. Они могут быть доступны на этапе компилляции, в класс-файлах, а также могут использоваться JVM во время исполнения программы. В Java 7 аннотации можно применять к пакетам, классам, методам, переменным и параметрам.

Как справедливо отмечают некоторые авторы, аннотации в том виде, в котором они реализованы в Java 7, не достаточно мощны для того, чтобы добавить поддержку контроля за изменяемостью объектов, так как в нынешней реализации нельзя аннотировать типы. Но уже в Java 8 такая поддержка появится, поэтому в данной работе именно аннотации используются для выражения неизменяемости объектов.

2.2 Система аннотаций

Будем считать, что каждая ссылка имеет модификатор изменяемости, который определяет, может ли быть изменено абстрактное состояние объекта, на который она ссылается. Этот моификатор определяется на уровне исходного кода, анализуруется на этапе компиляции и может иметь одно из четырех значений: @Mutable, @Immutable, @ReadOnly или @Isolated. На изображении ниже представлена иерархия параметров неизменяемости.

Выражение $A \preceq B$ будем трактовать как "A является наследником ". В данном случае, например, $Mutable \preceq ReadOnly$. Также будем считать,

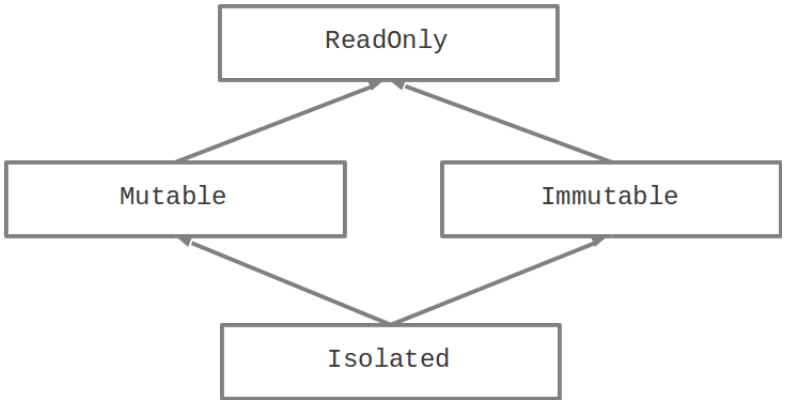

Рис. 2.1: Иерархия модификаторов неизменяемости

что если $A \preceq B$, где и B - модификаторы изменяемости, то $@A\,C \preceq @B\,C$, где - некий тип.

2.2.1 Ссылочная неизменяемость

Для поддержки ссылочной неизменяемости достаточно двух модификаторов: @Mutable и @ReadOnly. Состояние объекта не может быть изменено через @ReadOnly ссылку. Попытка присвоить поле через @ReadOnly ссылку или вызвать на ней меняющий объект метод приведет к ошибке компиляции:

Листинг 2.1: Mutable и RadOnly ссылки

```
1 @ReadOnly Person roPerson = ...;
2 String address = roPerson.address; // OK: reading field is
3                                    // always permitted
4 roPreson.address = "new address"; // Error: field can't be
5                         // assigned through ReadOnly referernce
6
7 @Mutable Person mPreson = ...;
8 mPerson.address = "new address"; // OK: mPerson is mutable,
9                                  // so field can be assigned
```

Пусть I(x) - это функция, которая принимает класс, тип или ссылку и возвращает ее модификатор изменяемости. Тогда вышеизложенное правило может быть написано следующим образом:

Правило 2.2.1. *o.someField = ... разрешено тогда и только тогда, когда* $I(o) = Mutable$

Изменяемая ссылка может быть передана везде, где ожидается неизменяемая ссылка. Таким образом, @Mutable Person является наследником @ReadOnly Person.

2.2.2 Аннотации на методах

В Java ключевое слово this внутри конструктора или нестатического метода является ссылкой на текущий объект. Изменяемость this зависит от контекста, а именно от метода, в котором появляется this. По умолчанию все методы изменяют объект, на котором вызываются. В таких методах this будет иметь модификатор @Mutable. Те методы, которые не изменяют объект, на котором они вызываются, должны быть помечены аннотацией @Const (по аналогии с C++), this в этих методах будет иметь модификатор @ReadOnly.

На @ReadOnly ссылках нельзя вызывать методы, которые меняют объект, на котором вызываются. Формально это правило может быть описано так:

Правило 2.2.2. *o.m(...) разрешено, если* $I(o) \preceq I(m)$*, где I(m) – модификатор изменяемости this в этом методе.*

Требуется, что $I(o) \preceq I(m)$ а не $I(o) = I(m)$ для того, чтобы через изменяемую ссылку можно было вызывать методы, не меняющие объект.

Рассмотрим на примере применение этих правил.

Листинг 2.2: Аннотации на методах

```
1 class Person {
2     String name;
3     @AsClass Date dateOfBirth;
4
```

```
5       public Person(String name, Date dateOfBirth) {
6           this.name = name;
7           this.dateOfBirth = dateOfBirth;
8       }
9
10      public void setName(String name) {
11          this.name = name;
12      }
13
14      @Const
15      public String getName() {
16          return name;
17      }
18
19      @AsClass
20      @Const
21      public Date getDateOfBirth() {
22          return dateOfBirth;
23      }
24
25      @Const
26      public boolean wasBornInYear(int year) {
27          return dateOfBirth.getYear() == year;
28      }
29
30      public void setYearOfBirth(int year) {
31          dateOfBirth.setYear(year);
32      }
33
34      public static void print(@ReadOnly Person person) {
35          ...
36      }
37  }
```

Присваивание this.name = name в 11 строке разрешено, так как $I(this) =$

$I(setName) = Mutable$, а согласно правилу 2.2.1 через @Mutable ссылку можно присваивать значение поля. Это присваивание было бы не разрешено, если бы оно было перемещено на 16 строку, так как this является @ReadOnly ссылкой в контексте метода getName. Вызов метода setYear на 31 строке разрешен согласно правилу 2.2.2, так как $I(dateOfBirth) = I(this) \preceq I(setyearOfBirth)$. Этот вызов метода не был бы разрешен на 27 строке, так как в контексте метода wasBornInYear $I(this) = ReadOnly$. Статический метод print на 34 строке принимает объект класса Person с любым модификатором изменяемости.

Поле dateOfBirth проаннотировано @AsClass. Это значит, что его модификатор изменяемости зависит от того, какой модификатор у this. Соответсвенно и результатом работы метода getDateOfBirth будет либо @Mutable ссылка (если сам он был вызван на объекте, доступном по @Mutable ссылке), либо @ReadOnly ссылка в противном случае:

Листинг 2.3: Использование аннотации AsClass

```
1 @ReadOnly Person roPerson = ...;
2 // OK: I(getYear) = ReadOnly
3 int year = roPerson.getDateOfBirth().getYear();
4 // OK: I(roPerson.getDateOfBirth()) = I(roPerson) = ReadOnly
5 roPerson.getDateOfBirth().setYear(2000);
6
7 @Mutable Person mPerson = ...;
8 // OK: I(mPerson.getDateOfBirth()) = I(mPerson) = Mutable
9 mPerson.getDateOfBirth().setYear(2000);
```

Аннотация AsClass может встречаться на полях метода, локальных переменных, возвращаемых значениях нестатических методов и параметрах методов.

2.2.3 Перегрузка методов

При перегрузке методов, метод класса-потомка должен оставить прежним или усилить модификатор неизменяемости, который имеет this в данном методе.

Правило 2.2.3. *Если метод m' перегружет метод m, то $I(m) \preceq I(m')$*

Например, метод класса-потомка может добавить аннотацию Const к перегружаемому методу, если ее не было в классе-предке, но не наоборот.

2.2.4 Объектная неизменяемость

Хотя @ReadOnly ссылки запрещают менять объект, на который ссылаются, никто не гарантирует, что этот объект не будет изменен при помощи какой-либо другой ссылки. Это хорошо иллюстрирует слудющий пример:

Листинг 2.4: Изменение объекта хранимого по @ReadOnly ссылке

```
1  @Mutable Person person = ...;
2  person.setYearOfBirth(2000);
3  @ReadOnly Person roPerson = person; // OK: @ReadOnly Person is
4                                     // supertype for
5                                     // @Mutable person
6  // 2000 will be printed
7  System.out.println(roPerson.getyearOfBirth());
8  person.setyearOfBirth(2013);
9  // 2013 will be printed
10 System.out.println(roPerson.getyearOfBirth());
11 }
```

При этом часто возникает ситуация, когда хочется не только гарантировать, что по данной ссылке нельзя менять объект, но и то, что данный объект вообще нельзя менять. Такие гарантии могут быть полезны, например, при многопоточном программировании – если про объект известно,

что он неизменяемый, то к нему можно безопасно обращаться из нескольких потоков без дополнительной синхронизации. Разработанная в данной работе система может давать такую гарантию: @Immutable ссылка всегда указывает на неизменяемый объект.

Листинг 2.5: @Mutable и @Immutable ссылки

```
1 @Mutable Person person = ...;
2 @Immutable Person iPerson = person; // Error: @Immutable
     Person is
3                                     // not supertype for
4                                     // @Mutable Person
5 @ReadOnly Person roPerson = iPerson; // OK: @ReadOnly Person is
6                                      // supertype for
7                                      // @Immutable Person
8 }
```

Из данного примера видна разница между @ReadOnly и @Immutable ссылками: если @ReadOnly ссылка может указывать как на изменяемый, так и на неизменяемый объект, то @Immutable ссылка всегда указывает только на неизменяемый объект.

2.2.5 Исключение полей из абстрактного состояния объекта

Одной из целей данной работы была разработка системы типов, которая бы давала гарантии относительно абстрактного состояния объектов, а не конкретной его реализации. Транзитивные гарантии неизменяемости для всех полей объекта в некоторых случаях могут быть слишком сильны. Например, поля, используемые для кэширования, часто не являются частью абстрактного состояния. Таким образом, необходим механизм, позволяющий исключать некоторые поля из абстрактного состояния объекта. В данной работе для этого используется аннотация @Transient, которая

обозначает, что данное поле не является частью абстрактного состояния.

Многие авторы ([2], [1]) разделяют два способа исключения поля из абстрактного состояния:

- Значение поля может быть переприсвоено даже через неизменяемую ссылку, но само значение в этом случае не может быть изменено.
- Значение поля не может быть переприсвоено, но при этом значение поля может быть изменено даже через @ReadOnly ссыллку.

Этот подход кажется несколько избыточным: такая тонкая настройка изменяемости нужна крайне редко и при этом приводит к некоторым проблемам в системе типов, которые приходится решать введением новых правил. Мы используем слеудющее правило: если поле помечено как @Transient, то его значение может быть присвоено вне зависимости от изменяемости объекта, при этом изменяемость самого значения регулируется дополнительной аннотацией (@ReadOnly, @Immutable или @Mutable).

2.2.6 Вложенные классы

Статические вложенные классы подчиняются всем тем же правилам, что и обычные классы. Нестатические вложенные классы имеют дополнительную ссылку на this (ссылка на экземпляр внешнего класса). Метод нестатического вложенного класса может быть объявлен как Const только если он не меняет обе ссылки this (свою и внешнего класса). Для неизменяемого объекта могут быть созданы только неизменяемые экземпляры его вложенных классов.

2.2.7 Неизменяемые классы

Существуют классы, все представители которых являются неизменяемыми объектами. Таковыми являются, например, java.lang.String и боль-

шинство потомков java.lang.Number. Обычно тот факт, что все представители некоего класса являются неизменяемыми, отражается в документации. Для этого разрешено использовать аннотацию @Immutable. Все методы класса, объявленного как @Immutable будут обрабатываться так, как будто они аннотированы как @Const.

2.2.8 Создание циклов неизменяемых объектов

Большинсвто неизменяемых объектов, тем не менее, модифицируются во время фазы их конструирования. Например, в неизменяемый список нужно сначала добавить все элементы. В этот момент список фактически будет меняться. Часто эта фаза локализуема непостредственно в конструкторе объекта – например, в конструктор неизменяемого списка может быть передан набор объектов, которыми этот список нужно заполнить, и после завершения конструктора объект уже можно по праву считать неизменяемым. Несмотря на то, что для большинства объектов фаза их создания заканчивается посе отработки конструктора, бывают случаи, когда такой подход неприменим. Одним из наиболее ярких примеров этого могут служить неизменяемые циклические структуры данных. Рассмотрим следующий пример:

Листинг 2.6: DoubleLinkedListNode.java

```
1 class DoubleLinkedListNode {
2     @AsClass DoubleLinkedListNode prev;
3     @AsClass DoubleLinkedListNode next;
4
5     @Immutable
6     public static DoubleLinkedListNode createTwoNodeList() {
7         // ???
8     }
9 }
```

Необходимо реализовать метод createTwoNodeList, который вернет неизменяемый двусвязный список из двух элементов. Это сделать не получится, так как соединять элементы списка друг с другом придется уже после создания. Можно, конечно, возвращать не @Immutable ссылку, а @ReadOnly:

Листинг 2.7: DoubleLinkedListNode.java

```
1  class DoubleLinkedListNode {
2      @AsClass DoubleLinkedListNode prev;
3      @AsClass DoubleLinkedListNode next;
4
5      @ReadOnly
6      public static DoubleLinkedListNode createTwoNodeList() {
7          @Mutable DoubleLinkedListNode n1 =
8                                  new DoubleLinkedListNode();
9          @Mutable DoubleLinkedListNode n2 =
10                                 new DoubleLinkedListNode();
11
12         n1.next = n2;
13         n2.prev = n1;
14
15         return n1;
16     }
17 }
```

Нетрудно видеть, что этом случае созданный список фактически будет неизменяемым, так как после завершения этапа создания, на него не останется ни одной @Mutable ссылки. Но этот момент необходимо было бы дополнительно отражать в документации, также результат работы этого метода не мог бы быть использован для передачи в метод, который требует именно @Immutable ссылку.

Ключевым моментом в объяснении того, почему именно результатом работы метода createTwoNodeList будет неизмяенемый объект было следующее утверждение: *после завершения этапа создания, на него не останется ни одной @Mutable ссылки.* На самом деле, важно еще и то, что @Mutable

ссылок не осталось и на другие транзитивно-достижимые из данного объекта объекты. И так как в данном случае это утверждение верно, то объект фактически является неизменяемым. Таким образом, можно прийти к определению *изолированной ссылки*:

Определение 2.2.1. *Изолированная ссылка (Isolated) – это ссылка на изолированный граф объектов. Объекты внутри изолиованного графа могут ссылаться друг на драга, но существует только одна внешняя не-ReadOnly ссылка на такой граф. Все пути к не @Immutable объектам, доступным через @Isolated ссылку, идут через это ссылку кроме путей, идущих по @ReadOnly ссылкам.*

Изолированная ссылка может быть единовременно конвертирована в @Mutable или @Immutable ссылку, так как на граф объектов, достижимых через нее, есть только @ReadOnly ссылки, которые не гарантируют ничего относительно этого графа.

Превращение @Isolated ссылки в @Mutable ссылку происходит тривиальным образом:

Листинг 2.8: Превращение @Isolated ссылки в @Mutable

```
1 @Isolated Person p = ...;
2 p.setName("Bob");
```

Здесь в строке 2 происходит неявное преобразование модификатора изменяемости p в @Mutable.

Isolated ссылка может быть также единовременно сконвертирована в @Immutable ссылку.

Листинг 2.9: Превращение @Isolated ссылки в @Immutable

```
1 @Isolated Person p = ...;
2 @Immutable imp = p;
3 p.setName("Alice"); // Error
```

Не смотря на то, что p была изначально объявлена как @Isolated, после присовения в imp она была приведена к @Immutable и объект, на который она ссылается, не может быть изменен.

Важный момент заключается в том, что превращение @Isolated ссылки в @Mutable не является необратимым. Например, следующий пример не содержит ошибок компиляции:

Листинг 2.10: Превращение @Isolated ссылки в @Mutable и обратно

```
1 @Isolated
2 public IntBox increment(Isolated IntBox b) {
3     b.value++;
4     return b;
5 }
```

В данном случае превращение ссылки обратно в @Isolated возможно, так как фактически ссылка b осталась изолированной. В работе [3] было сформулировано следующее правило, которое обуславливает, может ли @Mutable ссылка быть превращена обратно в @Isolated после выполнения некой операции:

Правило 2.2.4. *Если входной контекст выражения не содержит @Mutable ссылок, а выходной контекст выражения содержит одну @Mutable ссылку, то эта ссылка может быть превращена обратно в @Isolated.*

Действительно, в случае, когда язык запрещает иметь глобальные изменяемые значения, а также исключать поля из абстрактого состояния объекта, это правило работает. Если после проведения какой-либо операции появилась одна @Mutable ссылка, а перед началом операции ни одной @Mutable ссылки не существовало, то эта ссылка либо является ссылкой на объект, на который других ссылок не существует, либо на объект, который был только что создан.

Но запрет на существование глобальных изменяемых переменных является слишком сильным ограничением, так как в существующем коде уже

имеется большое количесвто подобных примеров. При этом очевидно, что существуют методы, которые никаким образом не взаимодействуют с глобальными изменяемыми переменными и полями, которые исключены из абстрактного состояния объекта. Будем называть такие методы чистыми и аннотировать их как @Pure. Тогда вышеприведенное правило применимо в контексте @Pure метода. Остальные типовые правила для @Isolated ссылок будут аналогичны тем, что приведены в [3].

Рассмотрим, каким образом введение @Isolated ссылок может решить проблему с созданием циклов неизменяемых объектов:

Листинг 2.11: DoubleLinkedListNode.java

```
1  class DoubleLinkedListNode {
2      @AsClass DoubleLinkedListNode prev;
3      @AsClass DoubleLinkedListNode next;
4
5      @Mutable
6      @Pure
7      private static CircularListNode doCreateTwoNodeList() {
8          @Mutable DoubleLinkedListNode n1 =
9                                       new DoubleLinkedListNode();
10         @Mutable DoubleLinkedListNode n2 =
11                                      new DoubleLinkedListNode();
12         n1.next = n2;
13         n2.prev = n1;
14         return n1;
15     }
16
17     @Immutable
18     public DoubleLinkedListNode createTwoNodeList() {
19         @Isolated DoubleLinkedListNode result =
20                                      doCreateTwoListNode();
21         return result;
22     }
23 }
```

2.3 Алгоритм вывода аннотаций

При разработке реальных приложений обычно используется большое количество библиотечного кода. Проаннотировать весь этот код вручную аннотациями неизменяемости не представляется возможным. Чтобы предложенную нами систему аннотаций можно было использовать в реальных приложениях, нужно обеспечить возможность работать с непроаннотированным кодом. Самое простое решение – это объявить, что все методы меняют объекты, на которых вызываются. Но тогда практически все объекты во вновь написанном коде (уже с использованием модификаторов неизменяемости) окажутся @Mutable.

Таким образом, необходимо разработать способ проаннотировать существующий код в автоматическом режиме. Далее представлено описание алгоритма, который позволяет вывести соответсвующие аннотации по байт-коду.

Пусть необходимо проаннотировать байт-код некой библиотеки. При этом, возможно, аннотации на некоторых методах или их параметрах уже известны (например, в документации явно написано, что все экземпляры некого класса являются неизменяемыми). Считается, что эти наперед данные аннотации проставлены правильно. Далее рассмотрены этапы аннотирования кода этой библиотеки.

2.3.1 Поля, не входящие в абстрактное состояние объекта

На первом этапе анализа все поля, объявленные как transient помечаются аннотацией @Transient. Все остальные поля помечаются как @AsClass.

2.3.2 Анализ методов на чистоту

На этом этапе необходимо проставить аннотации @Pure на тех методах, которые не взаимодействуют с глобальными @Mutable переменными и полями, помеченными как @Transient. Результатом работы алгоритма будет множество методов, которые можно пометить как @Pure.

Пусть M – множество всех аннотируемых методов, определим функцию $results : M \to \{Pure, NotPure, Unknown\}$, которая для каждого метода возвращает то, что на данном этапе известно о его чистоте:

- Pure – известно, что метод можно проаннотировать как @Pure.
- NotPure – известно, что метод нельзя проаннотировать как @Pure.
- Unknown – еще не известно, можно или нельзя проаннотировать метод как @Pure.

Тогда псевдокод алгоритма будет выглядеть следующим образом:

Листинг 2.12: Анализ чистоты методов

```
1  analyze(M, results)
2      prev = results;
3      while True
4          for m in M
5              if resluts[m] == Unknown
6                  results[m] = analyzeMethod(results, m)
7          if results == prev
8                  return M.filter(m -> (results(m) == Pure))
9          else
10             prev = results
```

То есть, методы анализирутся до тех пор, пока за очередную итерацию не станет известно ничего нового. По теореме о неподвижной точке данный алгоритм завершит свою работу, так как количество методов, про которые не известно, являются ли они @Pure не возрастает (если однажды было

вычислено значение функции results для некоторого метода m, отличное от Unknown, то оно уже больше никогда не будет изменено) и ограничена (количество методов не может быть меньше нуля).

Очевидно, что практически любая библиотека использует методы, не входящие в ее состав (например, методы из стандартной библиотеки). Пусть MExt – множество всех таких методов. Тогда определим функцию $ext : MExt \to \{Pure, NotPure\}$, которая возвращет Pure, если известно, что на методе стоит аннотация @Pure, а иначе возвращает NotPure.

Теперь рассмотрим, как должен быть устроен код функции analyzeMethod. При анализе метода на чистоту по очереди анализируются все инструкции байт-кода этого метода.

Листинг 2.13: Анализ чистоты методов

```
1  analyzeMethod(results, m) {
2      for insn in m.bytecode
3                  switch(insn)
4                      case PUTSTATIC:
5                          if (field_is_not_readonly) return
                               NotPure
6                      case GETSTATIC:
7                          if (field_is_not_readonly and
                               field_is_not_immutable)
8                              return NotPure
9                      case PUTFIELD:
10                         if (field_is_transient and
                               field_is_not_read_only)
11                     return NotPure
12                     case GETFIELD:
13                         if (field_is_transient and
                               field_is_not_readonly and
14                             field_is_not_immutable)
15                             return NotPure
16                     case INVOKEINTERFACE, INVOKESTATIC,
17                          INVOKESPECIAL, INVOKEVIRTUAL,
```

```
18                          result = m in M ? results(m) : ext(m)
19                          if (result != Pure) return result
20                      default:
21      return Pure;
22  }
```

При описаниии метода analyzeMethod, были использованы следующие флаги:

- field_is_not_readonly – возвращет true, если поле, ипользуемое на запись или на чтение текущей инструкцией байт-кода не помечено как @ReadOnly.
- field_is_not_immutable – возвращет true, если поле, ипользуемое на запись или на чтение текущей инструкцией байт-кода не помечено как @Immutable.
- field_is_transient – возвращет true, если поле, ипользуемое на запись или на чтение текущей инструкцией байт-кода помечено как @Transient.

2.3.3 Вычисление модификаторов изменяемости

Для каждого метода нужно вычислить селедующие модификаторы изменяемости:

- Аннотации на параметрах метода.
- Аннотация на возвращаемом значении.
- Для нестатических методов необходимо вычислить, какой модификатор имеет this в контексте этого метода.

Во всех трех случаях будем вычислять не непосредственно модификаторы неизменяемости, а те границы, в которых они могут лежать. При

вычислении этих модификаторов используется тот же самый подход, что и при вычислении чистоты методов:

Листинг 2.14: Анализ модификаторов изменяемости для методов

```
analyze(M, results)
    rpevResults = results
    while True
        for m in M
            results = analyzeMethod(M, results)
        if results == prevResults
            return results
        else
            revResults = results
```

Во время работы метода границы, вычисленные для каждого модификатора неизменяемости, могут либо сузиться, либо не измениться. Таким образом, гарантируется, что метод закончит свою работу.

Методы, в которых @ReadOnly будет попадать в границы, вычисленные для this, пометим как @Const. Для параметров методов выберем наиболее общий модификатор. Немного сложнее обстоит дело с вычислением модификатора на возвращаемом значении. Рассмотрим следующий метод:

Листинг 2.15: Вывод модификатора неизменяемости для возвращаемого методом значения

```
public Person getPerson() {
    if (...) {
        Peson tResult = createImmutablePerson();
        return tResult;
    } else {
        Person fResult = createMutablePerson();
        return fResult;
    }
}
```

Будем считать, что известно, что метод createImmutablePerson() возвращает @Immutable ссылку, а метод createMutablePerson() – @Mutable. Тогда $Immutable \preceq I(Result) \preceq ReadOnly$ и $Mutable \preceq I(tResult) \preceq ReadOnly$. Будем действовать следующим образом: сначла для каждого из возможных возвращаемых значений вычислим наиболее конкретный модификатор неизменяемости. Для tResult это будет @Immutable, а для fResult – @Mutable. После этого для полученных значений модификатора посчитаем максимально конкретный общий тип. В данном случае это будет @ReadOnly. Таким образом, возвращаемое значение метода getPerson() нудно пометить модификатором @ReadOnly.

Рассмотрим подробнее то, как именно будут вычисляться границы для ссылок в рамках одного метода. Этот анализ просиходит с помощью инструмента KAnnotator [1]. Строится граф потока данных. Далее на каждом ребре этого графа известно состояние стэка и текущие значения локальных переменных. Для каждой переменной хранится набор значений, которые она может иметь в данной точке графа. Для каждого из этих значений хранится интервал модификатора изменяемости. В момент, когда происходит ветвление графа, эти данные копируются для каждой из ветвей.

Когда происходит анализ одной ветви графа потока данных, поочередно анализируются инструкции байт-кода [2]. Некоторые из инструкций налагают дополнительные ограничения на ссылки, которые в данный момент находятся на стэке.

- INVOKESTATIC – для каждой ссылки, передаваемой в метод, ограничиваем сверху ее модификатор изменяемости согласно аннотации, стоящей на соответствующем параметре метода.

- INVOKEVIRTUAL, INVOKEINTERFACE, INVOKEDYNAMIC, INVOKESPECIAL – если вызываемый метод не помечен @Const, ограничиваем модифи-

[1] https://github.com/JetBrains/kannotator

[2] http://en.wikipedia.org/wiki/Java_bytecode_instruction_listings

катор изменяемости ссылки, на которой вызываем метод, модификатором @Mutable. С параметрами поступаем таким же образом, как и в случае со статическим методом.

- PUTFIELD – запись в поле объекта можно представить как вызов метода, не помеченного как @Const, принимающего единсвенный параметр, который проаннотирован так же, как и поле, в которое происходит присваивание.

- AASTORE, BASTORE, IASTORE, CASTORE, SASTORE, FASTORE, LASTORE, DASTORE – почмечаем массив как $Mutable \preceq I(array) \preceq Mutable$.

Также для каждой операции если какая-либо из ссылок, принимающих в ней участие является @Isolated, и передается туда, где ожидвается @Mutable ссылка, то проверяется, удовлетворяет ли эта операция устовию 2.2.4. В противном случае, на модификатор изменяемости для этой ссылки ставится @Mutable как ограничение снизу. Если какая-либо @Isolated ссылка была передана как @Immutable, то для этой ссылки @Immutable ставится как ограничение снизу.

Если в графе существует вершина, в которой сходятся несколько его ветвей, то необходимо слить данные, полученные на этих путях. Множества значений, полученные для одной переменной при этом объединяются, а границы модификаторов изменяемости для значений "сливаются"путем пересечения границ, вычисленных для каждого из значений на различных путях: так, если на одной из ветвей графа было получено, что $Mutable \preceq I(a) \preceq ReadOnly$ а другой – $Mutable \preceq I(a) \preceq Mutable$, то после слияния двух ветвей графа получим условие $Mutable \preceq I(a) \preceq Mutable$. Все варианты таких слияний представлены в таблице:

	(RO;RO)	(RO;Imm)	(RO;Isol)	(RO;Mut)	(Mut;Mut)	(Mut;Isol)	(Imm;Isol)	(Isol;Isol)	(Imm;Imm)
(RO;RO)	(RO;RO)								
(RO;Imm)	(RO;RO)	(RO;Imm)							
(RO;Isol)	(RO;RO)	(RO;Imm)	(RO;Isol)						
(RO;Mut)	(RO;RO)	(RO;RO)	(RO;Mut)	(RO;Mut)					
(Mut;Mut)	N/A	N/A	(Mut;Mut)	(Mut;Mut)	(Mut;Mut)				
(Mut;Isol)	N/A	N/A	(Mut;Isol)	(Mut;Mut)	(Mut;Mut)	(Mut;Isol)			
(Imm;Isol)	N/A	(Imm;Imm)	(Imm;Isol)	N/A	N/A	(Isol;Isol)	(Imml;Isol)		
(Isol;Isol)	N/A	(RO;RO)	(Isol;Isol)	N/A	N/A	(Isol;Isol)	(Isol;Isol)	(Isol;Isol)	
(Imm;Imm)	N/A	(Imm;Imm)	(Imm;Imm)	N/A	N/A	N/A	(Imm;Imm)	N/A	(Imm;Imm)

Из таблицы видно, что никакое слияние не расширяет границы для параметров.

В случае, когда в графе потока данных присутсвуют циклы, будем как бы "разворачивать"этот цикл и анализировать получающийся граф до тех пор, пока не будет достигнута неподвижная точка. Неподвижная точка будет достигнута, так как границы модификаторов неизменяемости для параметров при таком анализе могут только сужаться.

Во время работы данного алгоритма границы модификаторов изменяемости для всех значений могут только сужаться. Это гарантирует, что данный алгоритм обязательно закончит работу.

2.3.4 Неизменяемые классы

Если в процессе анализа оказалось, что какой-либо класс не имеет методов, не помеченных @Const, то данный класс можно автоматически пометить как @Immutable. При автоматическом аннотировании кода имеет смысл помечать как @Immutable только final классы, так как потомки класса могут добавить в интерфейс свои неконстантные методы, и, если бы класс был при этом помечен как @Immutable, добавление неконстантных методов в классах-потомках привело бы к ошибке компиляции.

2.4 Сравнение с существующими подходами

Данная работа имеет ряд особенностий по сравнению с уже существующими работами.

- В отличие от [1], в данной работе поддерживается не только объектная, но и ссылочная неизменяемость.

- В отличие от от IGJ([2]), была решена проблема с созданием неизменяемых циклических структур. Существует расширение языка IGJ, которое решает эту проблему – [4]. Но в этой работе вводится достаточно сложная концепция владения объектами (ownership), использование которой в значительной мере загромождает код. Также в нашей работе был разработан алгоритм автоматического вывода модификаторов неизменяемости для уже существующего кода, который отсутсвуте в [2] и [4], что затрудняет использование данных разработок в реальных проектах.

- В языке, преложенном в работе [3], не поддерживается исключение полей из абстрактного состояния объекта. Сам язык, при этом, не является в полной мере расширением C# – в нем отсутсвуют изменяемые глобальные переменные. Эти два ограничения в работе [3] необходимы, в том числе, для решения проблемы с созданием неизменяемых циклических структур. Нам удалось обойти эти ограничения. Также в [3] отсутствует способ использовать существующий код, написанный на C#.

3. Заключение

В данной работе предложена система аннотаций, позволяющая контролировать изменяемость объектов на этапе компиляции для языка Java. Данная система удовлетворяет следующим требованиям:

- Поддерживается как объектная, так и ссылочная неизменяемость.
- Поддреживается глубокая неизменяемость с возможностью исключать некоторые поля из абстрактного состояния объекта.
- Данная система работает на этапе компиляции и не влияет на поведение программы времени исполнения.
- Существует возможность "продлить"фазу конструирования неизменяемого объекта за пределы конструктора, что позволяет создавать неизменяемые циклические структуры.

Литература

1. Tschantz, M. S. (2006). Javari : Adding Reference Immutability to Java.

2. Zibin, Y., Potanin, A., Artzi, S., Kiezun, A., Ernst, M. D., Kie, A. (2007). Object and Reference Immutability using Java Generics.

3. Gordon, C. S., Parkinson, M. J., Parsons, J., Bromfield, A., Duffy, J. (2012). Uniqueness and reference immutability for safe parallelism. ACM SIGPLAN Notices.

4. Potanin, A., Ali, M., Ernst, M. D. (n.d.). Ownership and Immutability in Generic Java.

5. Boyapati, C., Lee, R. Rinard, M. (2002). Ownership Types for Safe Programming : Preventing Data Races and Deadlocks.

6. Pechtchanski, I., Sarkar, V. (2002). Immutability specification and its applications. Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande.

7. Birka, A. (2003). Compiler-Enforced Immutability for the Java Language.

8. Haack, C., Poll, E., Sch, J. (2005). Immutable Objects for a Java-like Language.

9. Haack, C., Poll, E. (2005). Type-based Object Immutability with Flexible Initialization.

10. Ma, K.-K., Foster, J. S. (2007). Inferring aliasing and encapsulation properties for java. ACM SIGPLAN Notices, 42(10), 423.

11. Luis, T., Jr, C., Ernst, M. D. (2007). Enforcing and Inferring Reference Immutability in Java.

12. Artzi, S., Kiezun, A., Glasser, D., Ernst, M. D., Kie, A. (2007). Combined Static and Dynamic Mutability Analysis.

13. Kjolstad, F., Dig, D., Acevedo, G., Snir, M. (2011). Transformation for class immutability. Proceeding of the 33rd international conference on Software engineering - ICSE '11, 61.