# 1. Введение

## 1.1 Статический анализ кода

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

Статический анализ позволяет выявить многие виды ошибок еще до запуска программы, большинство из которых сложно искать и воспроизводить непосредственно во время работы приложения. В связи с этим активно развиваются различные инструменты, позволяющие статически доказывать отсутсвие в программах ошибок тех или иных видов.

*привести примеры тулов в исторической последовательности*

## 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 представители которого являются неизменяемыми.*

Примером неизменяемого класса является, например, java.lang.String.

**Определение 1.2.2.** *Неизменяемый объект – объект, который не может быть изменен, при не гарантируется, что другие представители того же самого класса могут быть изменены.*

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

**Определение 1.2.3.** *неизменяемая ссылка – ссылка, которая не может быть использована для изменения объекта, на который она указывает (при этом объект может быть изменен через другую ссылку).*

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

Нужно заметить, что данные понятия не являются чем-то искусственным по отношению к языкам программирования. Приведем примеры использования данных понятий в языке программирования Java.

Например, в документации к классу org.joda.time.Period написано: "Неизменяемый временной период..."[1]. Таким образом, класс org.joda.time.Period является неизменяемым классом. *нужно ли приводить еще примеры*

## 1.3 Обзор существующих решений

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

### 1.3.1 C++

В языке C++ для выражения неизменяемости есть ключевое слово const.

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 reading,
10                            // not for writing
11
12         const S non_const_s;
13         non_const_s.val = 42;  // OK: non_const_s was not 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;
```

[1] http://joda-time.sourceforge.net/apidocs/org/joda/time/Period.html

```
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 the "pointee" data
15     ptrToConst  = NULL;  // OK: modifies the pointer
16
17     constPtr->val = 0; // OK: modifies the "pointee" data
18     constPtr  = NULL;  // Error: cannot modify the pointer
19
20     constPtrToConst->val = 0; // Error: cannot modify the "pointee" data
21     constPtrToConst  = NULL;  // Error: cannot modify the pointer
22 }
```

Теперь рассмотрим, что именно понимается под изменением какого-либо значения. 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. 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 на константных, так и на неконстантных объектах. Методы, непомеченные как const, могут быть вызваны только на неконстантных объектах.

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 is always mutable
13  }
```

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.

*добавить строгое определение глубокой неизменяемсти и картиночку*

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.

*ссылка на источник*

### 1.3.2 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 1.4: 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
10     MyClass mc = new MyClass();
11     mc.values = new int[100]; // Error: field values was declared final
12     mc.values[2] = 4; // OK: values is declared final, but we can still
13                       // change object, 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 1.5: 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
8   }
```

В данном случае, Collections.unmodifiableList(values) вернет обертку над исходным списком, у которой переопределены все изменяющие список методы 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 1.6: Использование неизменяемого списка

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

Таким образом, в стандратной библиотеке Java неизменяемые коллеции реализованы просто как обертки над стандартыми интерфесами, у которых переипределены меняющие 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 1.7: 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
8 }
```

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 [2]. В ней есть две отдельные иерархии - для изменяемых коллекций и для неизменяемых. Таким образом, попытка выхвать изменяющий коллекцию метод на неизменяемой коллекции приведет к ошибке компиляции.

Листинг 1.8: 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
8 }
```

С одной стороны, этот подход позволяет избежать ошибок, связанных с неправомерным изменением объектов во время выполнения программы, но с дургой он требует гораздо более тщательного продумывания интерфейсов, а также более трудоемок в реализации.

*добавить про документацию*

### 1.3.3 C#

В C# ключевое слово readonly, примененное к полям имеет тот же смысл, что слово final в Java: 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 1.9: Ключевое слово readonly

```
1  using System;
2  public class ReadOnlyTest
3  {
4      class MyClass
5      {
6          public int x;
7          public readonly int y = 25; // Initialize a readonly field
8          public readonly int z;
9
10         public MyClass()
11         {
```

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

```
12          z = 24;   // Initialize a readonly instance field
13      }
14
15      public MyClass(int p1, int p2, int p3)
16      {
17          x = p1;
18          y = p2;   // OK: readonly field can be reassigned in constructor
19          z = p3;
20      }
21  }
22
23  public static void Main()
24  {
25      MyClass p2 = new MyClass();
26      p2.x = 55;   // OK: field x is not readonly
27      p2.y = 33;   // Error: field y can't be reassigned, as it is readonly
28
29  }
30 }
```

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, объявленные как const всегда будет иметь одно и то же значение, так как являются константой времени компиляции.

Листинг 1.10: Ключевое слово const

```
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         {
11             z = 24;   // Initialize a readonly instance field
12         }
13
14         public MyClass(int p1, int p2)
15         {
```

```
16          x = p1;
17          y = p2;    // Error: const field can not be reassigned in
                constructor
18      }
19  }
20
21  public static void Main()
22  {
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 reassigned, as it is const
26
27  }
28 }
```

### 1.3.4 D

*нужно ли тут объяснить что вообще за язык такой - D? И если нужно, то не нужно ли то же самое делать про остальные яызки?*

Во второй версии языка прогаммирования D 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 1.11: const vs immutable

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

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.

Листинг 1.12: 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 immutable(Foo).
8                           // foo.next.num is of type immutable(int).
```

### 1.3.5 Javari

Javari - это расширение языка Java, которое добавляет в Java ссылочную неизменяемость, комбинируя статические и динамические проверки неизменяемости. Авторы вводят следующее определение:

**Определение 1.3.1.** *Австрактное 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 исключны из его абстракного состояния.

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

Если какая-либо ссылка объявлена как readonly, то она не может быть использована для изменения объекта, на который она указывает. 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 1.13: 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.

*рисуночек!*

На данном рисунке представлена иерархия классов в Javari. Система типов гарантирует, что изменяющие объект методы не могут быть вызваны на неизменяемых ссылках, и что объект, на который ссылается readonly переменная не может быть скопирован в не-readonly переменную.

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 1.14: 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 1.15: 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 1.16: 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 }
```

Javari также позволяет добавлять модификаторы мутабельности к типовым параметрам:

Листинг 1.17: Модификаторы мутабельности в типовых параметрах

```
1 /*mutable*/ List</*mutable*/ Date> ld1; // add/remove and mutate elements
```

```
2 /*mutable*/ List<readonly Date> ld2; // add/remove
3 readonly List</*mutable*/ Date> ld3; // mutate elements
4 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. 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. Например, 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 1.18: Объявление класса с типовыми параметрами

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

Также Javari позволяет абстрагироваться от изменяемости типового параметра. Фактически, ? readonly C – это укороченная запись для ? extends readonly C super mutable C. Так, List<? readonly Date> является суперклассом для List<readonly Date> и List<mutable Date>.

*рисуночек, а еще нужнодобавить про ололо безопасность пыщь пыщ*

Рассмотрим 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:

Листинг 1.19: Ключевое слово romaybe

```
1 class DateCell { Date value; romaybe Date getValue() romaybe { return
      value; }
2 }
```

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.

*плюсы, минусы, подводные камни подхода*

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:

- *некотнтролируемое утверждение* о том, что некая ссылка является неизменяемой: "x is mutable"
- *контролируемое утверждение* о том, что некая ссылка явлется измеянемой, если другая ссылка является изменяемой: "if y is mutable then x is mutable"

После составления системы утверждений алгоритм рашает ее. *нужно ли расписывать, как это решается?*

*плюсы, минусы, подводные камни алгоритма*

### 1.3.6 Immutability Generic Java

Immutablility Generic Java (IGJ) – 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 быть 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.

*еще тут много всяких ништяков + плюсы, минусы, подводные камни*

### 1.3.7 Uniqueness and Reference Immutability for Safe Parallelism

В работе Uniqueness and Reference Immutability for Safe Parallelism 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 также поддерживает полиморфизм относительно этих модификаторов, а также простое создание циклов неизменяемых объектов.

## 1.4 Постановка задачи

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

К данной системе были предъявлены следующие требования:

- Должна быть поддержана как объектная, так и ссылочная неизменяемость.
- Негобходима возможность исключать некоторые поля из абстрактного состояния объекта.
- Данная система должна давать возможность создавать неизменяемые циклические структуры объектов.
- Необходимо иметь вохможность использовать 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. Основная часть

## 2.1 Подход к технической реализации

Предположим, нужно добавить какую-то новую функциональность в язык. 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 изменению грамматики языка, из чего следует невозможность 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.

В случае Java есть несколько способов добавить поддержку неизменяемости объектов в язык. В работе IGJ это сделано с помощью добавления дополнитьельного типового параметра ко всем классам. Но это выглядит очень громоздко и трудно читаемо. Более того, информация о типовых параметрах остуствует в скомпилированных файлах, то есть информация об изменяемости также будет доступна только в исходном коде. Другой варинат – использование аннотаций.

Аннотация в Java – это вид синтаксических методанных, которые могут быть добавлены в исходный код. Они могут быть доступны на этапе компилляции, встроены в класс-файлы, 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

В данной работе каждая ссылка имеет модификатор изменяемости, который определяет, может ли быть изменено ее абстрактное состояние. 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 B". В данном случае, например, $Mutable \preceq ReadOnly$. Также будем считать, что если $A \preceq B$, где A и B - модификаторы изменяемости, то $@A \preceq @BC$, где C - 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 always permitted
3 roPreson.address = "new address"; // Error: field can't be assigned through
    ReadOnly referernce
4
5 @Mutable Person mPreson = ...;
6 mPerson.address = "new address"; // OK: mPerson is mutable, so field can be
    assigned
```

Пусть I(x) - это функция, которая принимает класс, тип или ссылку и возвращает ее модификатор изменяемости. Тогда формально 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.

*рисунок*

### 2.2.2 Аннотации на методах

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 строчке 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 строчку, так как this является ReadOnly ссфлкой в контексте метода getName. Вызов метода setYear на 31 строчке зарешен согласно правилу 2.2.2, так как $I(dateOfBirth) = I(this)I(setyearOfBirth)$. Этот вызов метода не был бы разрешен на 27 строчке, так как в контексте метода wasBornInYear $I(this) = ReadOnly$. Статический метод print на 34 строчке принимает объект класса Person с любым модификатором изменяемости.

Поле dateOfBirth проаннотировано AsClass. Это значит, что его модификатор изменяемости зависит от того, какой можификатор 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         int year = roPerson.getDateOfBirth().getYear(); // OK: I(getYear) =
              ReadOnly
3         roPerson.getDateOfBirth().setYear(2000); // OK:
              I(roPerson.getDateOfBirth()) = I(roPerson) = ReadOnly
4
5         @Mutable Person mPerson = ...;
6         mPerson.getDateOfBirth().setYear(2000); // OK:
              I(mPerson.getDateOfBirth()) = I(mPerson) = Mutable
7 }
```

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* $I(m)I(m')$

Например, метод класс-потомок 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
                supertype for @Mutable person
4           System.out.println(roPerson.getyearOfBirth()); // 2000 will be
                printed
5           person.setyearOfBirth(2013);
6           System.out.println(roPerson.getyearOfBirth()); // 2013 will be
                printed
7   }
```

При этом частно 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: Mutable ссылка указывает на изменяемый объект, а Immutable ссылка - на неизменяемый.

Листинг 2.5: Mutable и Immutable ссылки

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

Из данного примера видна разница между ReadOnly и Immutable ссымлками: 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 былы разработка системы типов, которая бы давала гарантии относительно абстрактного состояния объектов, а не о конкретной его реализации. Транзитивные гарантии неизменяемости для всех полей объекта в некоторых случаях могут быть слишком сильны. Например, поля, исползуемые для кэширования 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, которое обозначает, что данное поле не является частью абстрактного состояния.

Многие авторы 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 исходное значение 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, которые частно выглядят синтетически.

### 2.2.6 Вложенные классы

Статические вложенные классы подчинябтся всем тем же правилам, что и обычные классы. Нестатические вложенные классы имеют дополнительную ссылку на this. Изменяемость this зависит от того, в каком методе он вызван. Метод нестатического вложенного класса может быть объявлен как Const только если он не меняет обе ссылки this (свою и внешнего класса).

### 2.2.7 Неизменяемые классы

Существуют классы, все представители которых. Таковыми являются, например, java.lang.String и большинство потомков java.lang.Number. Обычно тот факт, что все представители некого класса являются неизменяемыми, отражается в документации. В данно работе для этого разрешено использовано 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 ни менее, модифицируются во время фазы их конструирования. Часто эта фаза локализуема непостредственно в конструкторе объекта – например, в конструктор неизменяемогосписка может быть пнабор объектов, которыми этот список нужно заполнить, и после отработки 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: CircularListNode.java

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

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

Листинг 2.7: CircularListNode.java

```
class CircularListNode {
    @AsClass CircularListNode prev;
    @AsClass CircularListNode next;

    @ReadOnly
    public static CircularListNode createTwoNodeList() {
        @Mutable CircularListNode n1 = new CircularListNode();
        @Mutable CircularListNode n2 = new CircularListNode();

        n1.next = n2;
        n1.prev = n2;
        n2.next = n1;
        n2.prev = n1;

        return n1;
    }
}
```

Не трудно 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 моментов 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 ссфлок 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-неизменяемым объектам, доступным через isolated ссылку идут через это ссылку кроме путей, идущих по ReadOnly ссылкам.*

*рисуночек*

Введение такого понятия, как 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 ссфлки в Mutable ссфлку происходит тревиальным образом:

Листинг 2.8: Превращение Isolated ссылки в Mutable

```
@Isolated Person p = ...;
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 изначальнообъявлена как Isolated, после присовения в imp она была преведена 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 *добавить ссылку* было сформулировано следующее правило, которое обуславливает, может ли Mutable ссылка быль 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, котоырй был только что создан.

Но запрет на существование глобальных изменяемых переменных кажется 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 вышепреведенное правило применимо в контексте Pure метода. Остальные типовые правила для Isolated ссылок будут аналогичны тем, что приведены в *ссылочка*

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

Листинг 2.11: CircularListNode.java

```
1  class CircularListNode {
2      @AsClass CircularListNode prev;
3      @AsClass CircularListNode next;
4
5      @Mutable
6      @Pure
7      private static CircularListNode doCreateTwoNodeList() {
8          @Mutable CircularListNode n1 = new CircularListNode();
9          @Mutable CircularListNode n2 = new CircularListNode();
10
11         n1.next = n2;
12         n1.prev = n2;
13         n2.next = n1;
14         n2.prev = n1;
15
16         return n1;
17     }
18
19     @Immutable
20     public CircularListNode createTwoNodeList() {
21         @Isolated CircularListNode result = doCreateTwoListNode();
22         return result;
23     }
24 }
```

## 2.3 Алгоритм вывода аннотаций

При разработке реальных приложений обычно используется большое количество библиотечного кода. Проаннотировать весь этот код аннотациями неизменяемости не представляется возможным. Чтобы разработанную в данной работе 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 необъходимо разработать способ проаннотировать существующий в автоматическом режиме. Далее представлено описание алгоритма, который теоретически позволяет с той или иной точностью вывести соответсвующие аннотации по байткоду.

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 данный аннотации проставлены правильно. Далее рассмотрены этапы аннотирования кода этой библиотеки.

### 2.3.1 Вычисление полей, не входящих 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 $results : M \to Pure, NotPure, Unknow$ которая для каждого метода возвращает то, что в 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

Определим также функцию $count : M, results \to int$ такую, что она возвращает количесвто методов m, для которых $results(m) = Unknown$. Тогда в упрощенном виде псевдокод алгоритма будет выглядеть следующим образом:

Листинг 2.12: Анализ чистоты методов

```
1  analyze(M, results)
2      prevUnknown = count(M, results);
3      while True
4          for m in M
5              if resluts[m] == Unknown
6                  results[m] = analyzeMethod(results, x)
7          c = count(M, results)
8          if c == prevUnknown
9                  return M.filter(\m.(results(m) == Pure))
10         else
11             revUnknown = c
```

То есть, методы анализирутся до тех пор, пока за очередную итерацию не станет известно ничего нового. По теореме о неподвижной точке (?) данный флгоритм завершит свою работу, так как функция count не возрастает (если однажды было вычислено значение функции reaults для некоторого метода m, отличное от Unknown, то оно уже больше никогда не будет изменено) и ограничена (количесвто 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.

- Если встречается инструкция PUTSTATIC и при этом не известно, что поле, в которое происходит запись, имеет модификатор изменяемости ReadOnly, то возвращается NotPure
- Если встречается инструкция GETSTATIC и при этом не известно, что поле, занчение которого считывается, имеет модификатор изменяемости Mutable, то возвращается NotPure
- Если встречается инструкция PUTFIELD и при этом поле, в которое происходит запись, проаннотировано как @Transient, то возвращается NotPure
- Если встручается инструкция GETFIELD и при этом поле, значение которого читается, является @Transient и @Mutable, то возвращается NotPure
- Если встречается инструкция вызова метода m, то вычисляется значение функции $result = m \in M?results(m) : ext(m)$. Если result != Pure, то врзвращается result
- Во всех остальных случаях возвращаем Pure

### 2.3.3 Вычисление модификаторов изменяемости

Для каждого метода нужно вычислить селедующие модификаторы неизменяемости:

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

При вычислении этих модификаторов используется тот же самый подход, что и при вычислении чистоты методов – то есть, алгоритм вычисления модификаторов неизменяемости является . Во всех трех случаях будем вычислять не непосредственно модификаторы неизменяемости, а те границы, в которых они могут лежать. После этого для параметров методов выберем наиболее общий модификатор, а для возвращаемого знаяения – наиболее конкретный. Те методы, в которых ReadOnly будет попадать в границы, вычисленные для this, пометим как @Const.

*добавить*

### 2.3.4 Вычисление неизменяемых классов

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 почемать 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

Несмотря на то, что существует множество работ, в той или иной степени предлагающих систему контроля за изменяемостью объектов для Java и других объектно-ориентированных языков, данная работа имеет несколько принципиальных особенностей.

- Во многих работах рассмотрела либо только объектная, либо только ссылочная неизменяемость. В данной работе сочетаются обе эти концепции.
- В отличае от *тыц* и *тыц*, решена проблема с созданием неизменяемых циклических структур. В отличае от *тыц* это удалось сделать без введения таких сложных концепций как владение объектов и без заметного утяжеления языка – если для всех классов, которые пишет конкретный программист фаза конструирования заканчивается в конструкторе, то потребовавшиеся изменения вообще не конснутся этих классов. Решение же, предложенное в работе *тыц*, было расширено на язык, содержащий изменяемые глобальные (статические) переменные и поля, исключенные из абстрактного состояния объекта.
- В данной работе также предложен алгоритм аннотирования существующего кода, отсутсвующий, например, в работах *тыц* и *тыц*, что позволяет проаннотировать существующий библиотечный код.

# 3. Заключение

## 3.1 Результаты работы

В данной работе удалось разработать систему, позволяющую контролировать изменяемость объектов на этапе компиляции. Данная система удовлетворяет всем необходимым требованиям к системе:

- Поддерживается как объектная, так и ссылочная неизменяемость.
- Поддредивается глубокая неизменяемость с возоможностью исключать некоторые поля из абстрактногосостояния объекта.
- Данная система полностью статическая и, как следствие, байт-код полученный в результате ее использования, может исполняться на обычной Java-машине.
- Существует возможность "продлить"фазу конструирования неизменяемого объекта за пределы конструктора.
- Разработан алгоритм, позволяющий проаннотировать существующий код с целью его дальнейшего использования уже из нового кода, создаваемого с использованием можификаторов неизменяемости.

## 3.2 Направления развития

# Литература


1. Кирилл Колышкин, Павел Емельянов, CRIU: больше, чем живая миграция для Linux контейнеров, Yet another Conference, 2012 год.

2. Jonathan Corbet, KS2010: Checkpoint/restart, lwn.net, 2 ноября 2010 год.

# Оглавление