УЧРЕЖДЕНИЕ РОССИЙСКОЙ АКАДЕМИИ НАУК

САНКТ-ПЕТЕРБУРГСКИЙ АКАДЕМИЧЕСКИЙ УНИВЕРСИТЕТ – НАУЧНО-ОБРАЗОВАТЕЛЬНЫЙ ЦЕНТР НАНОТЕХНОЛОГИЙ РАН

	На правах рукописи
	Диссертация допущена к защите Зав. кафедрой _____ “ ”_____2011 г.

Д И С С Е Р Т А Ц И Я

НА СОИСКАНИЕ УЧЕНОЙ СТЕПЕНИ

М А Г И С Т Р А

Тема: “Декларативное описание статического анализа программ на языке Java с аннотациями на типах”

Направление: 010600.68 – Прикладные математика и физика

Магистерская программа: "Математические и информационные технологии"

Выполнил студент С.С. Исакова
(подпись)

Руководитель: А.А. Бреслав
(подпись)

Рецензент: А.В. Смаль
(подпись)

Санкт-Петербург
2011 г.

Содержание

1. Введение

Параллельно с развитием языков программирования развиваются и автоматизированные методы обнаружения ошибок, допускаемых программистами. Многие из таких методов построены на статическом анализе программного кода [1-3]. Часто программисту предоставляется возможность аннотировать код дополнительной информацией, которая не требуется для преобразования программы в исполняемый код, но при этом позволяет производить более тонкий и точный статический анализ кода.

В версию 1.8 языка Java [4] планируется добавить поддержку аннотаций на типах [5]. Аннотации на типах как раз предоставляют возможность осуществлять анализ кода с учётом дополнительной информации от пользователя.

Расширение языка Java аннотациями на типах определяется в запросе на спецификацию JSR-308 [6]. Прототипом для этой спецификации является пакет Checker Framework [7-9]. В нём реализовано множество примеров аннотаций, и предоставляется механизм создания новых. Checker Framework подробно рассмотрен в разделе 2.2.

Существенным недостатком данного пакета является невозможность описывать семантику аннотаций декларативно. Декларативное описание позволяет сократить объем требуемого для написания кода, увеличить его читаемость, избежать множества ошибок [10-12]. Существует другой пакет JavaCOP [13], в котором поддерживается декларативный синтаксис задания семантики аннотаций (с некоторыми оговорками, подробнее описано в разделе 2.3.4). Однако в этом пакете не поддерживается расширение языка Java аннотациями на типах (такие аннотации необходимо записывать в комментариях), и он уступает в функциональности Checker Framework. Пакет JavaCOP подробно рассмотрен в разделе 2.3.

Целью данной работы является разработка и реализация языка декларативных описаний для аннотаций типов языка Java.

Основные задачи, решаемые в работе:

- разработка декларативного языка задания семантики аннотаций;
- реализация поддержки данного языка как расширение над Checker Framework;
- сравнение полученных результатов с существующими разработками.

2. Обзор средств описания семантики аннотаций на типах

В данной главе в разделе 2.1 рассматриваются предлагаемые изменения в синтаксисе языка Java посредством введения аннотаций на типах. Далее в разделе 2.2 рассказывается про пакет Checker Framework [7-9], являющийся прототипом спецификации JSR-308 [6]. В разделе 2.3 рассказывается про пакет JavaCOP [13], поддерживающий декларативный язык описания семантики аннотаций.

Стоит отметить, что здесь и далее в работе под семантикой аннотаций на типах понимается статическая семантика, а не семантика времени выполнения.

2.1. Синтаксис языка Java с аннотациями на типах

Аннотации введены в язык Java, начиная с версии 1.5. Они представляют собой некие метаданные, которые могут добавляться в исходный код программы и не влияют на ее выполнение, но могут использоваться компилятором, в том числе для статического анализа кода. В версии 1.5 языка Java аннотировать можно классы, поля, методы, переменные и т.д.

Примеры:

i. Для подавления предупреждений определенного типа можно использовать аннотацию `@SuppressWarnings`:

```
@SuppressWarnings("unchecked")
void myMethod() { ... }
```

ii. Аннотация `@Override` указывает на то, что аннотированный метод переопределяет метод суперкласса:

```
@Override
void mySuperMethod() { ... }
```

iii. Для тестового метода `m()` указывается аннотация `@Test`:

```
@Test public static void m() { ... }
```

В версию 1.8 языка Java планируется добавить возможность аннотировать типы. Это позволит ограничивать свойства переменных аннотированных типов и проделывать дополнительные проверки.

Примеры кода с аннотациями на типах:

i. Типом переменной `variable` является `@Nullable String`:

```
@Nullable String variable = null
```

Здесь стоит отметить, что использование аннотации может вызвать двусмысленность, например, в объявлении

```
@MyAnnotation String variable
```

аннотация `@MyAnnotation` может быть приписана как типу `String`, так и переменной `variable`. Чтобы предотвратить возможную неясность, при объявлении аннотации указывается, где она может использоваться. Например, следующим образом объявляется аннотация на типах:

```
@TypeQualifier
public @interface NonNull {
}
```

ii. При объявлении метода аннотируется тип возвращаемого значения и тип объекта, на котором метод может быть вызван:

```
@Immutable Object method() @Immutable
```

iii. Объявление класса с аннотированным типом параметра:

```
class MyClass<@Nullable T>
```

2.1.1. Расширения компилятора для обработки аннотаций

При компиляции [14] Java кода с аннотациями нужно запускать компилятор с определенными опциями, указывающими расширение (annotation processor) для обработки аннотаций. Компилятор Java принимает на вход пользовательский код с аннотациями, при этом за обработку аннотаций отвечает соответствующий Annotation Processor, который генерирует возможные ошибки и предупреждения (см.Рис. 2-1).

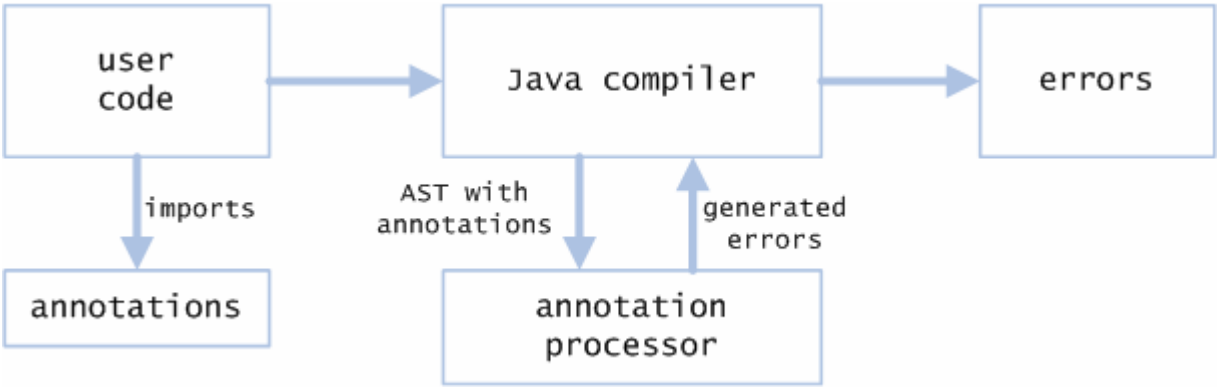


Рис. 2-1 Принцип работы расширений компилятора Java

AnnotationProcessor – это стандартный интерфейс, который нужно реализовать для расширения возможностей компилятора, в том числе для обработки аннотаций во время компиляции.

2.1.2. Семантика аннотаций @Nullable, @NonNull

Одним из основных видов ошибок при программировании на языке Java является попытка разыменования нулевой ссылки, которая приводит к исключению `NullPointerException`. Один из способов контролировать появление нулевых ссылок – это введение аннотаций на типах `@Nullable`, `@NonNull` [15-19].

Аннотация `@Nullable` на типе означает, что в переменной данного типа может храниться нулевое значение (и при разыменовании переменной мы должны проверить, что это не так). При попытке разыменования переменной, тип которой аннотирован `@Nullable`, возникает ошибка (или предупреждение) времени компиляции.

```
void method (@Nullable Object o) {
    o.toString();     //ошибка
    if (o != null) {
        o.toString(); //ok
    }
}
```

Аннотация `@NonNull` на типе означает, что в переменной данного типа не может храниться нулевого значения, т.е. при разыменовании гарантированно не случится `NullPointerException`.

```
void method (@NonNull Object o) {
    o.toString();  //ok
}
```

При этом компилятор следит, чтобы не было неявного приведения объекта аннотированного типа `@Nullable` к типу `@NonNull`. Например, в случае присваиваний (переменной типа `@NonNull` не может быть присвоен объект типа `@Nullable`), передаче аргументов методу, возвращении значения.

Таким образом, за счет вынуждения пользователя в некоторых местах добавлять проверки, гарантируется отсутствие исключения `NullPointerException` во время выполнения.

2.2. Checker Framework

Прототипом спецификации JSR-308, в которой описываются изменения языка Java при поддержке аннотаций на типах, является пакет Checker Framework. В нём реализованы многие аннотации на типах (такие как `@Nullable`, `@Immutable` – аналог ключевого слова `const` в C++), а также предусмотрена возможность определять новые аннотации. Далее описываются средства, необходимые для описания расширения компилятора (annotation processor), задающего семантику вводимых аннотаций.

2.2.1. Иерархия типовых аннотаций

Checker Framework позволяет декларативно (с помощью мета аннотаций) задавать иерархию на типовых аннотациях. После этого при присваиваниях, передаче аргументов

методу, возвращении значений проверяется, что фактический тип является подтипом ожидаемого типа в смысле иерархии типовых аннотаций.

Например, для аннотаций `@Nullable` и `@NonNull` иерархия аннотаций имеет вид:

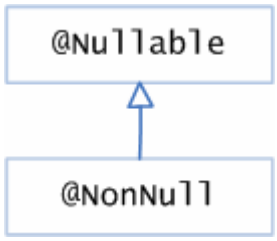


Рис. 2-2 Иерархия аннотаций

Для задания иерархии при объявлении аннотации указывается подтипом какой аннотации она является. Например, объявление аннотации `@NonNull`:

```
@SubtypeOf(Nullable.class)
public @interface NonNull {
}
```

При этом иерархия аннотированных типов выглядит следующим образом:

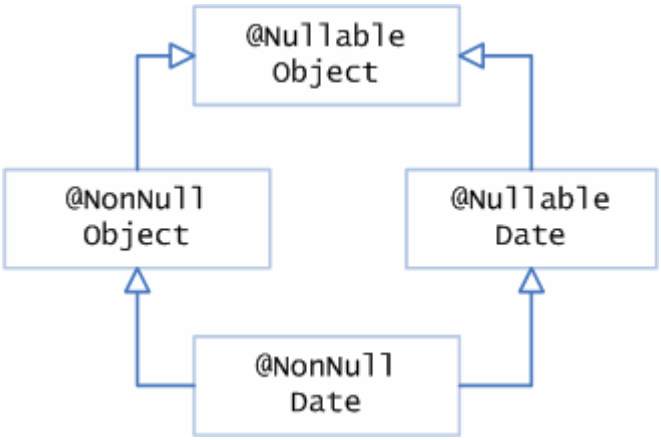


Рис. 2-3. Иерархия аннотированных типов

Здесь важно, что класс Date является наследником класса Object. Подобная иерархия будет поддерживаться для любых классов, связанных наследованием. Поддержка иерархии аннотированных типов осуществляется на стадии статического анализа кода при помощи расширений компилятора (Annotation Processor). Стоит отметить, что это не противоречит тому, что в Java нет множественного наследования (компилятор Java ничего не знает про аннотации, аннотации обрабатываются расширениями компилятора).

2.2.2. Неявное аннотирование

Также в Checker Framework поддерживается возможность аннотировать элементы по умолчанию (в простых случаях декларативно, с помощью мета аннотаций). Например,

когда мы хотим указать, что константа `null` должна по умолчанию аннотироваться как значение `@Nullable` типа, мы при объявлении аннотации указываем:

```
@ImplicitFor(trees = {Tree.Kind.NULL_LITERAL})
public @interface Nullable {
}
```

Соответственно, аннотацию `@NonNull` мы хотим по умолчанию приписать переменным и значениям примитивных типов и заведомо не нулевым ссылкам (созданным с помощью конструкции `new`):

```
@ImplicitFor(
    typeClasses={AnnotatedPrimitiveType.class},
    trees={
      Tree.Kind.NEW_CLASS,
      Tree.Kind.NEW_ARRAY,
      // All literals except NULL_LITERAL:
      Tree.Kind.BOOLEAN_LITERAL, Tree.Kind.CHAR_LITERAL,
      Tree.Kind.DOUBLE_LITERAL, Tree.Kind.FLOAT_LITERAL,
      Tree.Kind.INT_LITERAL, Tree.Kind.LONG_LITERAL,
      Tree.Kind.STRING_LITERAL
    })
  public @interface NonNull {  }
```

Более сложные случаи неявного аннотирования должны реализовываться на языке Java в соответствующих методах класса, наследуемого от `AnnotatedTypeFactory`.

2.2.3. Семантика использования аннотаций

Для определения семантики аннотаций нужно задать набор правил, нарушение которых приводит к ошибке. В Checker Framework возможные правила разделяются на два класса. Первый класс правил состоит из правил поддержки типовой иерархии (при присваиваниях, передаче аргументов методу и возвращении значений проверяется, что неявное приведение типов происходит в соответствии с заданной иерархией). Эти правила реализованы в классе `BaseTypeVisitor`.

Второй класс правил определяет специфическое для аннотаций поведение. Например, в случае `@Nullable` аннотаций определяется, что разыменовывать `@Nullable` ссылку можно только после предварительной проверки. В Checker Framework данная логика записывается на Java коде при помощи переопределения соответствующих методов класса `BaseTypeVisitor`.

Например, правило о разыменовании нулевых ссылок записывается при обходе соответствующего поддерева: `visitMemberSelect(MemberSelectTree node, …)`. При

этом логика правила записывается императивно, т.е. перечислением последовательности действий, которые нужно совершить. Далее приведен код, отвечающий в Checker Framework за логику “сгенерировать ошибку при разыменовании нулевой ссылки”.

```
@Override
public Void visitMemberSelect(MemberSelectTree node, Void p) {
  if (!TreeUtils.isSelfAccess(node))
    checkForNullability(node.getExpression(), "dereference.of.nullable");
  return super.visitMemberSelect(node, p);
}
private void checkForNullability(ExpressionTree tree,
                                 @CompilerMessageKey String errMsg) {
  AnnotatedTypeMirror type = atypeFactory.getAnnotatedType(tree);
  if (!type.hasAnnotation(NONNULL))
    checker.report(Result.failure(errMsg, tree), tree);
}
```

Весь дополнительный анализ (например, сравнение переменной с константой `null` в условии) происходит ранее, при вызове `atypeFactory.getAnnotatedType(tree)` возвращается аннотированный тип уже с учётом предварительного анализа.

Цель данной работы – предоставить возможность программисту записать это правило в декларативном стиле. То есть придумать способ связать исходные данные (разыменование непроверенной `@Nullable` переменной) и результат (предупреждение компилятора о возможном исключении `NullPointerException`). В принципе, в Checker Framework именно это и делается, однако разглядеть по сути декларативное правило в императивной записи становится уже не так просто.

2.3. JavaCOP

В этом разделе рассматривается другое средство описания семантики аннотаций, в котором поддерживается декларативный синтаксис, – JavaCOP.

2.3.1. Декларативное описание семантики аннотаций

Код задания семантики аннотаций в JavaCOP состоит из правил и объявлений (ограничений):

```
rule ruleName(TreeType t){
    require(...):
        error(...);
}
declare defName(TreeType t){
    require(...);
}
```

Правило – это возможность задать некоторые ограничения на вершину конкретного типа, при невыполнении которых генерируется ошибка. Объявление – это в первом приближении просто именованное ограничение. На самом деле при помощи одноименных объявлений можно объединять ограничения, накладываемые на вершины разных типов.

Например, мы заводим правило, в котором используется ограничение `defNotNull`:

```
rule checkFieldAccessSafe(JCFieldAccess fa){
  require(defNotNull(fa.selected)):
    error(...);
}
```

При этом соответствующее ограничение определяется отдельно для вершин разных типов:

```
declare defNotNull(JCAssign a) { ... }
declare defNotNull(JCParens p) { ... }
declare defNotNull(JCMethodInvocation a) { ... }
declare defNotNull(JCConditional c) { ... }
```

Рассмотренное выше правило “сгенерировать ошибку при разыменовании нулевой ссылки” выглядит следующим образом:

```
rule checkFieldAccessSafe(JCFieldAccess fa){
  require(defNotNull(fa.selected) ||
          (fa.getSymbol != null && fa.getSymbol.isStatic)):
    error(fa, "Dereference of possibly null object.");
}
```

Здесь требуется, чтобы при доступе к полю (разыменовании объекта) выполнялось ограничение `defNotNull` для объекта. В противном случае будет сгенерирована ошибка "Dereference of possibly null object".

2.3.2. Сравнение с Checker Framework

В Checker Framework предусмотрена возможность декларативно определять иерархию аннотаций и неявно аннотировать значения (было описано в разделах 2.2.1, 2.2.2). В JavaCOP аналогичная функциональность должна описываться с помощью объявления ограничений:

```
declare defNotNull(JCTree t){
  require(t instanceof JCNewClass || t instanceof JCNewArray);
}
declare defNotNull(JCLiteral t){
  require(t != null && t.value != null);
}
declare defNotNull(JCTree t){
  require(t.type.isPrimitive());
```

```
}
```

Здесь мы определяем, что выделенные с помощью ключевого слова `new` ссылки, значения и переменные примитивных типов удовлетворяют ограничению `defNotNull`. Та же самая логика в Checker Framework записывается следующим более наглядным образом:

```
@ImplicitFor(
  typeClasses={AnnotatedPrimitiveType.class},
  trees={
    Tree.Kind.NEW_CLASS,
    Tree.Kind.NEW_ARRAY,
    // All literals except NULL_LITERAL:
    . . .
    })
public @interface NonNull {  }
```

То есть объявление некоторых ограничений в JavaCOP можно было бы упростить.

2.3.3. “Технические” ограничения

В JavaCOP нужно определять некоторое количество “технических” ограничений. Пример такого ограничения, которое не несёт специального для семантики нулевых аннотаций смысла:

```
declare defNotNull(JCParens p){
  require(defNotNull(p.expr));
}
```

Здесь мы определяем, что выражение в скобках обладает теми же ограничениями, что и это же выражение без скобок.

Возвращаясь к целям данной работы – хотелось бы количество декларативных правил (ограничений) минимизировать, оставив необходимость писать только те, которые соответствуют семантике статического анализа, а технические ограничения “спрятать” от пользователя.

2.3.4. Правила для условий

В JavaCOP для написания правил для условий не поддерживается декларативный синтаксис. Требуется описывать данную логику на Java коде, переопределяя класс `AbstractFlowFacts`.

Например, для поддержки семантики нулевых аннотаций в классе `NonNullFlowFacts` производится некоторый анализ для поиска сравнений переменных со значением `null`,

после чего результаты этого анализа учитываются при объявлении ограничения `defNotNull`:

```
declare defNotNull(Ident id){
  require(NonNullFlowFacts nff;
          nff <- id#getFlowFacts(NonNullFlowFacts#class)) {
    require(nff#isIdentNonnull(id));
  }
}
```

В предлагаемом в работе языке правила для условий записываются декларативно, аналогично остальным правилам.

3. Декларативный язык задания семантики аннотаций

В данной работе предлагается декларативный язык для задания семантики аннотаций на типах. Код на предлагаемом языке состоит из правил. Правила бывают двух типов: генерация фактов и проверка фактов.

Применение правил происходит следующим образом: обходится абстрактное синтаксическое дерево (AST), в каждой его вершине для каждого правила проверяется, можно ли применить это правило для поддерева данной вершины. Если какое-то правило применимо для поддерева вершины, то, соответственно, либо генерируются новые факты, либо происходит проверка. В случае если проверка прошла не успешно, генерируется ошибка.

Далее всё это рассматривается более подробно. В разделе 3.1 описываются типы правил и синтаксис языка. В разделе 3.2 описывается логика применения правил.

3.1. Синтаксис языка правил

В разделе 3.1.1 рассказывается про отношения (основную сущность предлагаемого языка). В разделе 3.1.2 рассматривается язык Scala, на основе которого реализован предлагаемый язык правил, в том числе рассказывается о такой возможности Scala, как сопоставление с образцом. Далее в разделе 3.1.3 подробно описывается синтаксис правил. В разделе 3.1.4 объясняется, почему порядок применения правил должен быть недетерминированным.

3.1.1. Отношения

Предлагаемый язык построен на генерации и проверке фактов про отношения. Основным аргументом отношения является элемент анализа кода:

- поддерево абстрактного синтаксического дерева, либо
- элемент потока выполнения (переменная, метод).

Поддерживаются нульместные, одноместные и двухместные отношения. Для одноместных отношений аргумент – это всегда элемент анализа кода. У двуместных отношений два аргумента: элемент анализа кода и значение пользовательского типа.

Рассмотрим конкретные примеры отношений. Для анализа нулевых ссылок вводится одноместное отношение “точно не нулевая ссылка”:

```
val notNull = UnaryRelation()
```

Аргументом такого отношения может быть и переменная как элемент потока выполнения, и конкретное использование переменной (поддерево AST).

Для анализа, находящего бессмысленные сравнения и присваивания, определяется отношение, связывающее переменную со своим значением:

```
val varValue = Function[VariableValue]()
```

В данном случае отношение представляет собой функцию, так как каждой переменной соответствует только одно значение. При объявлении отношения мы указываем пользовательский тип, здесь это `VariableValue`. Элементами данного типа будут либо значения примитивных типов, либо значения типа “ссылка”, либо специальный маркер “значение неизвестно”.

3.1.2. Сопоставление с образцом

Предлагаемый язык является встроенным доменно-специфичным языком на языке Scala [20-22]. Используется такая возможность языка Scala, как сопоставление с образцом (pattern matching) [23, 24].

На языке Java можно написать конструкцию сравнения с одним из возможных значений. Например, если `Day` – соответствующее перечисление, а `day` – переменная типа `Day`, то мы можем написать:

```
switch(day) {
  case MONDAY:  doSmth(day);
  case TUESDAY: doSmthElse(day);
  ...
}
```

Сопоставление с образцом – это обобщение приведенной выше конструкции. В Java в качестве аргумента `switch` могут использоваться только значения примитивных типов, строки и элементы перечислений. В качестве обобщения хотелось бы иметь возможность сравнивать элементы с более сложными шаблонами.

Синтаксис такого сравнения на Scala:

```
expression match {
  case Pattern1 => Action1
  case Pattern2 => Action2
  ...
}
```

Рассмотрим конкретный пример шаблона. Пусть на Scala заведены специальным образом классы `Inequality`, `Expression`, `Variable`, `Literal`. При этом у класса `Inequality` есть единственный конструктор с двумя параметрами типа `Expression`, а классы `Variable` и `Literal` – наследники `Expression`. У конструкторов классов `Variable` и `Literal` по одному параметру (переменная как элемент потока выполнения и примитивное значение, соответственно).

Тогда может быть создан объект (предполагается, что переменная `i` доступна):

```
val ineq = Inequlity(Variable(i), Literal(42))
```

Рассмотрим шаблоны, с которыми мы можем пытаться сравнить объект `ineq`:

```
ineq match {
  case Inequality(Variable(v), Variable(u)) => ...
  case Inequality(Variable(v), Literal(null)) => ...
  case Literal(“procrastination”) => ...
  case Inequality(Literal(l), _) => ...
  case Inequality(left, right) => ...
  case _ => ...
}
```

Подчеркивание _ заменяет универсальный шаблон, который применим ко всему. В рассмотренном примере объект `ineq` удовлетворяет последнему и предпоследнему шаблону.

В шаблоне могут объявляться переменные. Тогда при сопоставлении в переменные будут записаны конкретные значения. Например, в шаблоне

```
Inequality(left, right)
```

объявляются переменные `left` и `right`. В рассмотренном выше примере после сопоставления в переменной `left` будет содержаться `Variable(i)`, в переменной `right` – `Literal(42)`.

При написании правил в предлагаемом языке будет использоваться сравнение с образцом, при этом сравниваемым объектом будет поддерево абстрактного синтаксического дерева, соответствующее какому-то фрагменту кода.

Предположим, происходит анализ кода:

```
s != null
```

В этом случае соответствующее абстрактное синтаксическое дерево выглядит примерно следующим образом:

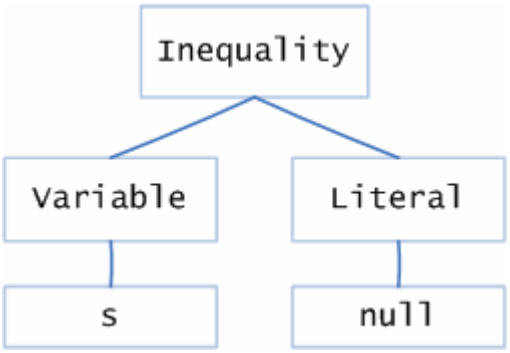


Рис. 3-1. Фрагмент абстрактного синтаксического дерева

Правило, которое должно применяться для всех случаев сравнения переменной с литералом `null`, будет содержать сопоставление с шаблоном:

```
case Inequality(Variable(v), Literal(null)) => ...
```

Например, при анализе кода

```
array[0] != null
```

соответствующее поддерево AST уже не пройдет сопоставление с шаблоном, так как в правой части сравнения не переменная, а доступ к элементу массива.

В языке Scala конструкция `case` представляет собой частичную функцию (подробнее можно прочитать, например, в [20, глава 15]) и её можно использовать не только в конструкции `match`. Запомнив объявленную частичную функцию, в дальнейшем можно проверить, применима ли данная функция к аргументу, и если да, вычислить значение функции от аргумента.

Например, можно сохранить в переменной `partialFunction` следующую частичную функцию:

```
val partialFunction = { case Variable(v) => v }
```

Это функция, которая может быть применена только к объектам типа `Variable`. Применение частичной функции к объектам может осуществляться следующим образом:

```
if (partialFunction.isDefinedAt(t))
  partialFunction.apply(t)
```

Здесь вначале мы проверяем, что частичная функция определена на объекте `t`, и если это так, вычисляем результат применения функции к `t`.

В предлагаемом в данной работе языке правила представляют собой частичные функции.

3.1.3. Типы правил

Код на предлагаемом языке – это множество правил. Правила могут быть трех типов:

- генерация фактов про отношения,
- генерация фактов при анализе условий,
- проверка фактов про отношения.

Анализ фрагмента кода в условии отличается от анализа этого же кода, например, при инициализации переменной:

```
if (s != null) { ... }
boolean b = (s != null);
```

В первом случае при анализе конструкции `if` информация про выполнение и невыполнение условия `(s != null)` должна быть доступна в дальнейшем соответственно при анализе тел `then` и `else` веток. Во втором случае условие `(s != null)` не влияет на дальнейший анализ.

Поэтому, выделяется отдельный тип правил: правила, которые применяются, если происходит анализ условия.

Для правил проверки фактов есть два подтипа:

- требование наличия фактов: если при проверке требуемых фактов нет, генерируется ошибка (см. пример iii);
- проверка наличия фактов: если проверяемые факты были сгенерированы, происходит ошибка (см. пример iv).

На Рис. 3-2представлены типы правил.

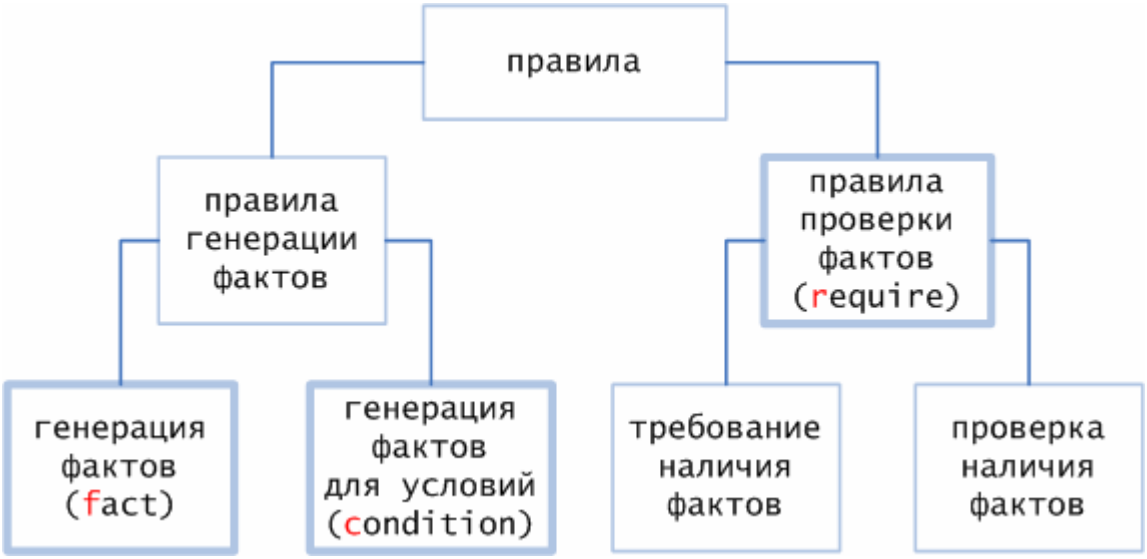


Рис. 3-2. Типы правил в предлагаемом языке

Все правила синтаксически выглядят похожим образом (курсивом выделены необязательные параметры):

```
f {case Pattern => fact {where (UserRelation)} (UserRelation)}
c {case Pattern => fact {where (UserRelation)} (UserRelation)}
r {case Pattern => require {where (UserRelation)} (UserRelation)
                           (ErrorMessage) (PostAction)}
r {case Pattern => check {where (UserRelation)} (UserRelation)
                         (ErrorMessage) (PostAction)}
```

Вначале определяется тип правила:

- **f** – генерация факта (**f**act),
- **c** – генерация факта для условия (**c**ondition),
- **r** – проверка факта (**r**equire)).

На месте `Pattern` пишется шаблон, с которым будут сравниваться поддеревья абстрактного синтаксического дерева при попытке применить правило.

На месте `UserRelation`, соответственно, записывается отношение, для которого генерируется или проверяется факт. Можно использовать ключевое слово `not` для отрицания отношения (для нульместных и одноместных отношений).

Конструкция

```
{where (UserRelation)}
```

является необязательной и ограничивает область применения правила.

Для случаев проверки фактов:

- `ErrorMessage` – сообщение об ошибке, которое выводится, если требуемый факт отсутствует;
- `PostAction` – действие, которое нужно сделать после проверки. Является необязательным параметром, может быть одной из двух констант: `POST_FACT` и `POST_NEG_FACT`. Первая константа означает, что нужно сгенерировать факт `UserRelation`, вторая – что нужно сгенерировать обратный факт (применяется только для нульместных и одноместных отношений).

Языка Scala позволяет при описании шаблона задавать дополнительные условия:

```
case Pattern if (conds) => ...
```

Объект будет удовлетворять шаблону только при выполнении этих условий. Подобная возможность также используется при написании правил (см. пример v).

Примеры.

i. Генерация фактов.

```
f {case VarChange(x, NewClass(_)) =>
        fact (varValue(x, Reference())}
```

Смысл правила в следующем: если мы встречаем утверждение вида

```
x = new SomeClass(arguments);
```

мы генерируем факт о том, что переменная x связана отношением varValue со значением Reference(). Так как отношение varValue – это функция, то данный факт означает, что значением переменной x является новая ссылка.

ii. Генерация фактов для условий.

```
c {case Inequality(Var(v), Literal(null)) =>
        fact (notNull(v))}
```

Если мы встречаем сравнение в условии:

```
if (v != null) {
…
}
```

мы генерируем факт, что для переменной v выполняется факт notNull(v).

iii. Проверка обязательных фактов.

```
r {case FieldAccess(obj, member) =>
        require (notNull(obj))
        ("Dereference of nullable") (POST_FACT)}
```

В анализе нулевых ссылок при разыменовании объекта мы проверяем, что для разыменовываемого объекта имеется факт, показывающий, что этот объект ненулевой. Это факт мог быть получен, если объект – это переменная, объявленная с типом, аннотированным @NonNull (включая неявно аннотированные переменные), либо если это переменная, про которую мы предварительно проверили, что она ненулевая. Соответственно, при проверке мы требуем наличия факта notNull(obj), если его не оказывается, мы генерируем ошибку с текстом "Dereference of nullable".

После этого мы запоминаем информацию о том, что объект obj ненулевой (если obj – это переменная, запоминаем, что она ненулевая). Это нужно для уменьшения количества однотипных ошибок. Тем более, если при разыменовании переменной не случилось исключения, то дальше её безопасно разыменовывать.

iv. Проверка необязательных фактов.

```
r {case Equality(v, u) =>
        check {varValue(v) `==` varValue(u)}
        ("Senseless comparison")}
```

Мы отслеживаем ситуацию, когда сравниваются переменные, про значения которых известно, что они совпадают. Точнее, переменные ссылаются на один и

тот же объект или в них записано одно и то же примитивное значение. Такие сравнения в программе редки, и они свидетельствуют о вероятной ошибке.

В проверке `check` как бы проверяется утверждение “А вдруг у нас достаточно информации, чтобы сказать, что отношение верно”.

Отличие проверки `check` от проверки `require` состоит в следующем:

- во-первых, генерируется ошибка при выполнении (`UserRelation`), а не при невыполнении, как в предыдущем случае;
- во-вторых, если информация про запрашиваемые элементы отсутствует, ошибка не генерируется.

Например, в приведенном выше примере, если про значения `varValue(v)` и `varValue(u)` в нашем анализе ничего простого не известно (что верно в большинстве случаев), то ошибки нет.

v. Пример правила с использованием конструкции `where` и дополнительных условий на шаблон.

Если переменная не была аннотирована явно, ее тип (`@NonNull` или `@Nullable`) будет зависеть от присваиваемых значений. Например, при присваивании не аннотированной переменной ненулевого значения, она становится ненулевой:

```
String s = “not null”;
s.toString();           //no warning
s = null;               //warning
```

Однако если мы объявляем переменную как `@Nullable`, присваивание ей ненулевого значения не должно менять тип переменной:

```
@Nullable String s = “not null”;
s.toString();           //warning
s = null;               //no warning
```

Здесь на шаблон, который проверяется при попытке применить правило, накладывается дополнительное условие: переменная не должна быть аннотирована значением `@Nullable`. При этом соответствующий факт генерируется только в том случае, когда про присваиваемое выражение известно (есть раннее сгенерированный факт), что оно ненулевое.

```
f {case Assignment(Var(v), expr)
            if !hasAnnotation(v, "Nullable") =>
        fact { where (notNull(expr)) } (notNull(v))}
```

Конструкция `where` предоставляет возможность записывать условия с использованием ранее сгенерированных фактов.

3.1.4. Порядок применения правил

Порядок применения правил недетерминирован. Это обусловлено теоретическим требованием к декларативности языка. В декларативном языке от порядка вычислений ничего не должно зависеть. В данном случае результат выполнения статического анализа не должен зависеть от порядка правил.

Конечно, ничего не запрещает записать два правила такого типа:

```
h { case Pattern(element) => notNull(element) }
h { case Pattern(element) => not (notNull(element)) }
```

Это пример неправильных правил, противоречащих друг другу. В практической реализации при детерминированном порядке применения правил противоречие не будет заметно, так как результат последнего выполненного правила удалит результат предыдущего. При недетерминированном порядке результаты для статического анализа будут меняться при повторных запусках, что позволит понять, что какие-то правила противоречат друг другу.

3.2. Применение правил

В разделе 3.2.1 кратко описывается теория абстрактной интерпретации (теоретической основы для обоснования корректности используемой аппроксимации). Далее в разделах 3.2.2-3.2.6 описывается статическая аппроксимация семантики языка Java.

3.2.1. Абстрактная интерпретация

Абстрактная интерпретация [25-28] – это теория, формализующая аппроксимацию семантики программ. Она была введена в работе [25] в 1977 году как некоторое обобщение методов статического анализа программного кода.

Каждой точке программы ставится в соответствие абстрактное состояние, представляющее собой надмножество всех возможных в этой точке конкретных состояний. Множество абстрактных состояний называется абстрактным доменом и должно образовывать решетку [29-31].

В процессе статического анализа перебираются вершины графа потока управления. Для каждой вершины вычисляются:

- входное состояние как объединение выходных состояний предыдущих вершин;
- выходное состояние как применение связанного с вершиной оператора к ее входному состоянию.

Процесс повторяется до тех пор, пока состояния не перестанут изменяться.

Существует теорема Кнастера - Тарского о том, что монотонный оператор на полной решетке имеет наименьшую неподвижную точку [32]. При этом неподвижная точка может быть получена путем многократного применения оператора к наименьшему элементу решетки (так как решетка полная, то в ней нет бесконечных цепей [29], и этот процесс конечный).

Из этой теоремы следует, что при реализации абстрактного интерпретатора для того, чтобы анализ гарантированно завершал свою работу, требуется выполнение двух условий:

- решётка абстрактных состояний должна быть полной (для этого достаточно показать, что в ней нет быть бесконечных цепей [27]);
- оператор, определяемый для каждой вершины на решётке должен быть монотонным.

3.2.2. Аппроксимация семантики языка Java

В данной работе механизм для проведения статического анализа (абстрактный интерпретатор) реализован как расширение компилятора (см. раздел 2.1.1). Ему на вход подаётся абстрактное синтаксическое дерево, а на выходе должен быть список сообщений об ошибках. Абстрактный интерпретатор в процессе выполнения использует правила для записи конкретных статических анализов. Это изображено на Рис. 3-3.

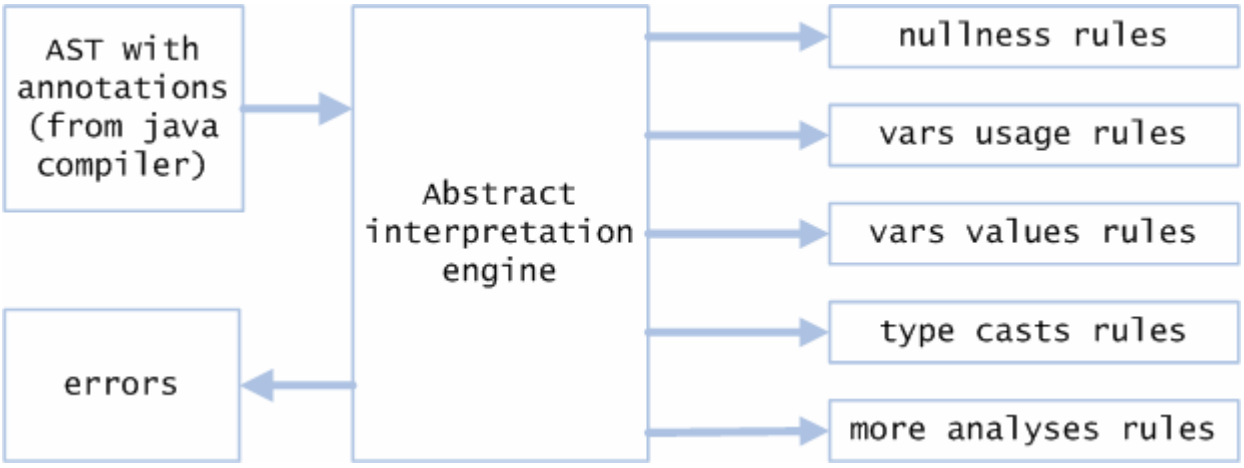


Рис. 3-3. Схема работы абстрактного интерпретатора

Множество хранимых фактов образует абстрактное состояние. Абстрактный интерпретатор применяет правила (изменяет состояние) в процессе обхода абстрактного синтаксического дерева.

Каждой вершине соответствует оператор, который изменяет входное состояние для данной вершины. Применение этого оператора заключается в том, что отбираются

правила, которые могут быть применены к соответствующему вершине поддереву. Если применимо правило генерации фактов, абстрактное состояние изменяется, в него добавляются новые факты. Если применимо правило проверки фактов, происходит соответствующая проверка. В этом случае при отсутствии требуемых фактов генерируется ошибка.

Если вершина абстрактного синтаксического дерева – это вход в цикл, происходит многократное применение правил генерации фактов (с пересечением результата с предыдущим состоянием) до достижения неподвижной точки. В остальных случаях правила применяются один раз.

Далее определим порядок на множестве абстрактных состояний. После этого покажем, что оператор, определяемый для каждой вершины на решётке, является монотонным относительно заданного порядка. Тогда для доказательства того, что абстрактный интерпретатор завершает свою работу, для каждого конкретного анализа надо будет показать, что решетка абстрактных состояний для этого анализа не имеет бесконечных цепей.

Вначале зададим порядок на фактах про отношения, а также операции объединения и пересечения фактов.

Определение.

- Если `ZeroRelation` – нульместное отношение, то будем говорить, что факт `ZeroRelation` является строгим уточнением факта `not (ZeroRelation)`:
  `ZeroRelation() < not (ZeroRelation).`

- Если `UnaryRelation` – одноместное отношение, то будем говорить, что факт $fact_1$`(e) = UnaryRelation(e)` является строгим уточнением факта $fact_2$`(e) = not (UnaryRelation(e))` для любого аргумента (напомним, что аргументом может быть только элемент анализа кода):
  `UnaryRelation(e) < not (UnaryRelation(e)).`

- Если `BinaryRelation[Value]` – двуместное отношение, являющееся функцией, то будем говорить, что факт $fact_1$`(e) = BinaryRelation(e, sub)` является строгим уточнением факта $fact_2$`(e) = BinaryRelation(e, sup)`:
  `BinaryRelation(e, sub) < BinaryRelation(e, sup)`
  если для элементов пользовательского типа задан соответствующий порядок:
  `sub < sup.`

- Если `BinaryRelation[Value]` – двуместное отношение, не являющееся функцией, то будем говорить, что факт, состоящий из множества отношений $fact_1(e)$ = `{BinaryRelaion(e, sub[i])}` является строгим уточнением факта $fact_2(e)$ = `{BinaryRelaion(e, sup[j])}`:

  `{BinaryRelation(e, sub[i])} < {BinaryRelation(e, sup[j])}`

  если множество элементов `{sub[i]}` является подмножеством `{sup[j]}`.

Утверждение. Рефлексивное замыкание (отношение “являться уточнением”) заданного на множестве фактов отношения “являться строгим уточнением” является порядком.

Определение.

- Для нульместных и одноместных отношений пересечением двух фактов про отношения является меньший факт, а объединением – больший.
- Для двуместного отношения, являющегося функцией, и фактов

  $fact_1(e)$ = `BinaryRelation(e, v)`

  $fact_2(e)$ = `BinaryRelation(e, u)`

  пересечением фактов является факт:

  `fact(e)` = `BinaryRelation(e, v` $\cap$ `u),`

  а объединением:

  `fact(e)` = `BinaryRelation(e, v` $\cup$ `u).`

  Соответственно, для элементов пользовательского типа должны быть заданы операции объединения и пересечения.
- Для двуместного отношения, не являющегося функцией, при пересечении и объединении фактов:

  $fact_1(e)$ = `{BinaryRelaion(e, v[i])}`

  $fact_2(e)$ = `{BinaryRelaion(e, u[j])}`

  пересекаются и объединяются, соответственно, множества элементов `{v[i]}` и `{u[j]}`.

Теперь можно задать порядок на множестве абстрактных состояний.

Определение. Состояние `s` является уточнением состояния `t`:

$s \le t$

если для каждого элемента анализа кода e, для которого присутствует факт в `s` ($fact_s(e)$), присутствует факт в `t` ($fact_t(e)$), при этом факт в `s` является уточнением факта в `t`:

$fact_s(e) \le fact_t(e)$.

Утверждение. Заданное на множестве состояний отношение “являться уточнением” является порядком.

Введём операции пересечения и объединения состояний.

Определение. Пересечение состояний s и t происходит следующим образом:

- для каждого элемента анализа кода, для которого существуют факты в обоих состояниях ($fact_s(e)$, $fact_t(e)$) в пересечение состояний добавляется пересечение этих фактов;
- для каждого элемента анализа кода, для которого только в одном из состояний существует факт, в пересечение состояний не добавляется фактов про этот элемент.

Определение. Объединение состояний s и t происходит следующим образом:

- для каждого элемента анализа кода, для которого существуют факты в обоих состояниях ($fact_s(e)$, $fact_t(e)$) в объединение состояний добавляется объединение этих фактов;
- для каждого элемента анализа кода, для которого только в одном из состояний существует факт, в объединение состояний добавляется этот существующий факт про элемент.

Утверждение. Для операций пересечения и объединения состояний выполняются следующие неравенства:

$$s \cap t \leq s \leq s \cup t,$$
$$s \cap t \leq t \leq s \cup t.$$

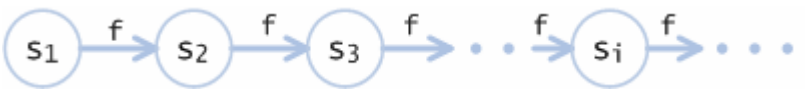
Утверждение. Множество состояний является решёткой относительно заданного на них порядка.

■ Для того чтобы показать, что множество состояний является решеткой, надо показать, что для любых двух состояний существуют точная верхняя и нижняя границы. Из предыдущего утверждения следует, что состояния $s \cap t$ и $s \cup t$ являются нижней и верхней границей соответственно. □

Утверждение. Для любой вершины оператор, ей соответствующий, является монотонным относительно заданного на множестве состояний порядка.

■ Если вершина не является входом в цикл, то область определения оператора – это единственное состояние (входное состояние для вершины), и он является монотонным.

Если вершина является входом в цикл, то при многократном применении оператора ко входному для данной вершины состоянию получается цепь, являющаяся подрешёткой решётки абстрактных состояний:



Покажем это, то есть покажем, что $s \leq f(s)$.

Применение оператора, соответствующего вершине v, состоит в следующем:

- пусть вход для данного оператора – это состояние s;
- вначале применяются правила, которые могут быть применены к вершине v, в процессе этого состояние s изменяется (добавляются новые факты, которые могут заменить уже существующие), в результате получается состояние t;
- выходом является объединение состояний s и t.

Так как множество абстрактных состояний является решёткой, выполняется:

$$s \leq s \cup t = f(s).$$

Область определения оператора – множество состояний $\{s_i\}$. На состояниях определен порядок: $s_1 \leq s_2 \leq s_3 \ldots \leq s_i \ldots$

Условие монотонности оператора записывается следующим образом:

$$s_i \leq s_j \Leftrightarrow f(s_i) \leq f(s_j)$$

Последнее неравенство эквивалентно следующему (так как $f(s_i) = s_{i+1}$):

$$f(s_i) \leq f(s_j) \Leftrightarrow s_{i+1} \leq s_{j+1}$$

Мы получили, что оператор монотонный, если верно:

$$s_i \leq s_j \Leftrightarrow s_{i+1} \leq s_{j+1}$$

Последняя эквивалентность всегда выполняется в силу заданного порядка. Поэтому, оператор монотонный. □

Мы определили порядок на множестве абстрактных состояний и показали, что оператор, определяемый для каждой вершины на решётке, является монотонным относительно заданного порядка. Теперь для доказательства того, что абстрактный интерпретатор завершает свою работу, для каждого конкретного анализа надо будет показать, что решетка абстрактных состояний для этого анализа не имеет бесконечных цепей.

3.2.3. Семантика конструкторов и методов

Для данной работы представляет интерес статический анализ кода, в котором в каждой вершине анализируется только доступная в этой вершине информация. Например, при анализе метода доступны тело метода и глобальные объявления вне метода. Операции, производимые внутри других методов, не являются доступными.

Поддержка доступа к телу другого метода сильно усложняет анализ. А одно из требований к статическому анализу – чтобы он был максимально простым.

Исключение частично составляют конструкторы. В случае объявления полей класса часто инициализация поля происходит далее в конструкторе, а в неинициализированном состоянии в переменную записывается значение `null`.

В анализе нулевых ссылок подобную ситуацию хотелось бы обрабатывать корректно:

```
1  class MyClass {
2      @NonNull Object o1;          //warning
3      Object o2;
4      Object o3;
5      public MyClass() {
6          o2 = new Object();
7      }
8      void method() {
9          o1.toString();
10         o2.toString();
11         o3.toString();           //warning
12     }
13 }
```

В приведенном примере объект `o1` некорректно инициализирован (так как в конструкторе нет инициализации ненулевым значением), объекты `o2` и `o3` инициализированы корректно, однако по умолчанию аннотируются разными аннотациями. Объект `o2` аннотируется по умолчанию как значение `@NonNull` типа (так как в конструкторе есть инициализация ненулевым значением), а объект `o3` – как значение `@Nullable` типа (инициализация отсутствует и в `o3` хранится `null`).

В результате статического анализа приведенного фрагмента кода выводится два предупреждения о возможных ошибках: при объявлении переменной `o1` (в строке 2) и при разыменовании переменной `o3` (в строке 11).

Для поддержки описанной функциональности при объявлении полей классов происходит анализ конструкторов и статических инициализаторов, и результат анализа (состояние) запоминается. При наличии нескольких конструкторов (и статических

инициализаторов) после их анализа результатом является объединение соответствующих состояний.

3.2.4. Семантика условий

При анализе условий применяются дополнительно правила для условий. Условия встречаются в конструкциях `if-then-else` и циклах.

Для анализа нулевых ссылок основное правило следующее: при сравнении переменной с константой `null` сгенерировать факт о том, что переменная ненулевая.

На предлагаемом языке это правило записывается следующим образом:

```
c {case Inequality(Var(v), Literal(null)) => fact (notNull(v))}
```

Одного этого правила достаточно, чтобы корректно обрабатывать не только условие, в котором сразу проверяется неравенство:

```
if (s != null) { ... }
```

но также и более сложные условия, в которых эта проверка не единственная, или она “спрятана” за отрицанием:

```
if (s != null && ...) { ... }
if (!(s == null)) { ... }
if (s == null) { ... } else { ... }
```

В последнем фрагменте кода в `else`-ветке должна быть информация о том, что переменная `s` ненулевая.

На самом деле при написании правила для условия в шаблоне (в примере выше `Inequality`) неявно кроется два шаблона: для обычного случая и для случая, когда происходит анализ условия в отрицании.

Пусть мы проверяем, удовлетворяет ли поддерево абстрактного синтаксического дерева `tree` шаблону `Inequality`:

- в обычном случае в шаблоне `Inequality` проверяется, что `tree` – это бинарное выражение с оператором “неравенство”;
- в случае анализа условия в отрицании в шаблоне `Inequality` проверяется, что `tree` – это бинарное выражение с оператором “равенство”.

Аналогично для других шаблонов, используемых в правилах для условий.

Для анализа логических операторов требуется поддержка операций пересечения и объединения состояний (см. раздел 3.2.2). Например, если объединяются два состояния, при этом в первом состоянии присутствует факт `notNull(v)`, а во втором отрицание `not (notNull(v))`, то в результат должен быть сохранен более общий факт `not (notNull(v))`.

Рассмотрим пример объединения состояний, в которых есть факты про двуместные отношения. Например, пусть определено двуместное отношение `varValue`. Пусть в первом состоянии записан факт `varValue(v, Reference())`, а во втором `varValue(v, Primitive(5))`. В этом случае в итоговом состоянии должно оказаться `varValue(v, Undefined)`, потому что `Undefined` – это объединение значений `Reference()` и `Primitive(5)`. Но если бы отношение `varValue` не было объявлено как функция, в итоговом состоянии оказалось бы `varValue(v, Set(Reference(),Primitive(5)))`.

Анализ логических операторов происходит следующим образом:

- `&&` (конъюнкция с ленивым вычислением аргументов). Анализируется левая часть условия, полученный результат используется при анализе правой части. Результатом является состояние, полученное при анализе правой части условия.
- `||` (дизъюнкция с ленивым вычислением аргументов). Анализируется левая часть условия. После этого анализируется левая часть условия в случае отрицания, полученный результат используется при анализе правой части. Результатом является состояние – объединение полученных в результате анализа левой и правой частей состояний.
- `&` (конъюнкция). Анализируются левая и правая части условия, после чего полученные в результате состояния пересекаются.
- `|` (дизъюнкция). Анализируются левая и правая части условия, после чего полученные в результате состояния объединяются.

При анализе конструкции `if-then-else` условие анализируется дважды: для обычного случая и для случая отрицания. После этого полученное в первом случае состояние используется при анализе блока `then`, а состояние, полученное при анализе условия в случае отрицания, используется при анализе блока `else`. Для окончательного результата состояния `then` и `else` блоков объединяются.

3.2.5. Семантика циклов

Обход циклов осуществляется несколько раз до стабилизации множества получаемых фактов. Практически указывается константное число раз, при достижении которого считается, что обход зациклился и его стоит прервать с соответствующим сообщением об ошибке.

Рассмотрим подробнее обход цикла `while`:

```
while (condition) {
```

```
  body
}
```

До начала обхода есть какое-то накопленное к этому моменту множество фактов, хранящихся в состоянии $state_0$. Результатом (результирующим состоянием) анализа цикла в том случае, если условие `condition` не верно и тело цикла ни разу не выполнится, будет это состояние:

$$result_0 = state_0$$

После первого обхода условия и тела цикла множество фактов могло поменяться и теперь это другое состояние ($state_1$). Текущим результатом анализа цикла (при условии, что тело цикла будет выполняться не более одного раза) является объединение этих состояний:

$$result_1 = result_0.\ union(state_1)$$

Далее аналогичным образом мы получаем результат после двух обходов цикла ($state_2$), в этом случае результат анализа цикла:

$$result_2 = result_1.\ union(state_2)$$

Такая процедура будет продолжаться, пока текущий результат анализа цикла

$$result_i.\ union(state_{i+1})$$

не стабилизируется. При осуществлении статического анализа неизвестно, сколько раз выполнится тело цикла, поэтому действительными считаются только те факты, которые генерируются вне зависимости от количества раз обхода.

Пока при анализе цикла применялись только правила для генерации фактов (ожидая пока множество генерируемых фактов не стабилизируется). Правила для проверки фактов применяются только при последнем обходе (практически после стабилизации доменов делается еще один обход для применения этих правил), чтобы исключить дублирование сообщений об ошибках.

Рассмотрим пример обхода цикла `while`:

```
String s = “not null”;
while (s.length() < 10) {   //error
  s = null;
}
```

Проследим изменения состояния в процессе анализа цикла:

- до начала анализа цикла состояние состоит из одного факта:

  $$result_0 = state_0 = \{\ notNull(s)\ \}$$

- при анализе условия новых фактов не добавляется; при анализе тела цикла добавляется новый факт `not (notNull(s))`, который заменяет предыдущий:

  $$state_1 = \{\ not\ (notNull(s))\ \}$$

- при объединении состояний в результате остаётся более общий факт:

  $result_1$ = $result_0$. union($state_1$) = { not (notNull(s)) }
- при последующем обходе цикла снова генерируется факт not (notNull(s)), который не меняет состояние:

  $result_2$ = { not (notNull(s)) }

Множество фактов стабилизировалось и осуществляется последний обход условия и тела цикла, в процессе которого применяются правила для генерации фактов, и выдается предупреждение об ошибке при разыменовании переменной s в условии цикла (так как в состоянии нет информации о том, что переменная s ненулевая).

Обход остальных циклов (do-while, for, foreach) происходит аналогичным образом.

3.2.6. Семантика конструкций throw, return, break

Присутствие или отсутствие одной из конструкций throw, return, break в теле конструкций if-then-else и циклов может изменить результат анализа.

Например, результат анализа конструкции if-then-else в следующем примере совпадает с результатом анализа условия в случае отрицания:

```
if (...) {
  ...
  return ...;
}
```

В данном примере последующий код написан будто бы в else ветке.

Другой пример: если анализируется цикл while, в теле которого нет конструкции break, то после анализа цикла должно выполняться отрицание условия.

Для поддержки подобных случаев запоминается информация о том, что встретилась соответствующая конструкция throw, return или break.

4. Примеры статического анализа, реализованного на предложенном языке

На предлагаемом языке реализован упоминавшийся ранее (см. раздел 2.1.2) анализ нулевых ссылок, а также несколько достаточно простых примеров статического анализа, не использующих аннотации на типах, однако дающих возможность оценить мощь и удобство предлагаемого языка.

4.1. Анализ нулевых ссылок

Семантика анализа нулевых ссылок была описана в разделе 2.1.2. Далее приводятся правила, необходимые для реализации анализа на предлагаемом языке.

4.1.1. Правила

Первую группу составляют правила неявного аннотирования переменных, которые объявлены с не аннотированным типом. Например, при анализе фрагментов кода:

```
String s1 = null;
String s2 = “meow”;
```

переменная `s1` должна аннотироваться как значение аннотированного `@Nullable` типа, а переменная `s2` – как значение аннотированного `@NonNull` типа.

Подобные правила записываются для объявления переменных и присваиваний переменным, что объединено в шаблоне `VarChange`:

```
f {case VarChange(v, expr)
            if (expr != null) &&
               !v.`type`.isPrimitive &&
               !hasAnnotation(v, "Nullable") =>
            fact { where (notNull(expr)) } (notNull(v))}

f {case VarChange(v, expr)
            if (expr != null) &&
               !v.`type`.isPrimitive &&
               !hasAnnotation(v, "NonNull") =>
            fact { where (not (notNull(expr))) } (not(notNull(v)))}
```

Следующее правило отслеживает объявления переменных аннотированных `@NonNull` типом, которые не проинициализированы в конструкторе:

```
class MyClass {
    @NonNull Object o;
    MyClass() {}
}
```

Как было описано в разделе 3.2.3, анализируются тела конструкторов, после чего применяются правила к объявлениям переменных. В следующем правиле при объявлении переменной требуем выполнения факта про эту переменную (этот факт должен был появиться при анализе тела конструктора):

```
r {case VarDecl(v, init)
            if (init != null) &&
               hasAnnotation(v, “NonNull”) =>
            require (notNull(v)) (“Uninitialized nonnull value”)}
```

При обращении к статическому полю или статическому методу класса не может случиться NullPointerException:

```
Pattern.compile("\\d");
System.out.println("hello!");
```

Поэтому, происходит неявное аннотирование использований классов как значений @NonNull типа. Во втором случае также не хотелось бы получать предупреждение о возможной ошибке при обращении к переменной System.out (хотя в этом случае не гарантируется отсутствие исключения, например, мы можем переопределить поток вывода по умолчанию), но для удобства предупреждения при обращении к полям класса System не генерируются:

```
f {case cr @ ClassRef(clazz) =>
            fact (notNull(cr))}
f {case ms @ MemberSelect(Ident(s, "System"), member) =>
            fact (notNull(ms))}
```

При приведении к типу свойства "ненулевая ссылка" и "возможно, нулевая ссылка" сохраняются:

```
void method(@NonNull Object o1, @Nullable Object o2) {
    @NonNull String s1 = (String) o1;
    @Nullable String s2 = (String) o2;
}
```

На предлагаемом языке правил это записывается следующим образом:

```
f {case tc @ TypeCast(expr, clazz) =>
            fact {where (notNull(expr))} (notNull(tc))}
f {case tc @ TypeCast(expr, clazz) =>
            fact {where (not(notNull(expr)))} (not(notNull(tc)))
```

Далее описываются правила генерации фактов для условий.

При сравнении переменной с константой null, нужно сгенерировать соответствующий факт:

```
c {case Inequality(Var(v), Literal(null)) =>
            fact(notNull(v))}
c {case Inequality(Literal(null), Var(v)) =>
            fact(notNull(v))}
```

Если переменная сравнивается на равенство с другой переменной, про которую известно, что она ненулевая, то при выполнении условия свойство "быть ненулевой ссылкой" распространяется и на первую переменную:

```
void method(Object v, @NonNull Object u) {
    if (v == u) {
        v.toString();   //no error
```

```
        }
    }
```

С помощью правил приведенную выше логику можно записать так:

```
c {case Equality(Var(v), Var(u)) =>
            fact { where (notNull(v)) } (notNull(u))}
c {case Equality(Var(v), Var(u)) =>
            fact { where (notNull(u)) } (notNull(v))}
```

Условие

```
if (v instanceOf ClassName) { ... }
```

в Java будет выполняться, только если значение `v` не `null`. Поэтому, при осуществлении подобной проверки мы можем запоминать факт о том, что переменная `v` ненулевая:

```
c {case InstanceOf(Var(v), clazz) =>
            fact (notNull(v))}
```

Главное правило анализа нулевых ссылок: при разыменовании переменной у нас должна быть информация, что она ненулевая.

```
r {case FieldAccess(obj, member) =>
            require (notNull(obj))
                    ("dereference of nullable")
                    (POST_FACT)}
```

После разыменования повторные предупреждения о разыменовании этой же переменной можно не генерировать (если случается исключение, то оно случается при первом обращении к переменной). Для поддержки этой функциональности используется ключевое слово `POST_FACT`. В данном примере после применения правила будет сгенерирован факт `notNull(obj)`.

В сокращенном цикле `for` происходит неявное разыменование коллекции, поэтому следует проверять, что коллекция ненулевая.

```
    void method(Collection<String> collection) {
      for (String s : collection) {
        s.toString();
      }
    }
```

Записываем соответствующее правило:

```
r {case EnhancedForLoop(el, collection) =>
            require (notNull(collection))
                    ("dereference of nullable collection")
                    (POST_FACT)}
```

При доступе к элементу массива переменная, в которой хранится массив, не должна быть нулевой.

```
String[] array = null;
String s = array[0];   //NullPointerException
```

Правило:

```
r {case ArrayAccess(array, index) =>
            require (notNull(array))
                    ("accessing nullable")
                    (POST_FACT)}
```

В следующей группе правил отслеживается ситуация неявного приведения завернутого целочисленного (`Integer`, `Long`, …) или булева (`Boolean`) типа к примитивному типу (`int`, `long`, `boolean`, …). Здесь нет явного разымевания переменной, однако также может случиться исключение.

```
Integer i = null;
int j = i + 5;      //NullPointerException
String s = i + “”;  //ok
```

При этом в случае конкатенации строк приведение к примитивному типу не производится. Для поддержки описанной логики используются следующие шаблоны:

- `ArithOp`. Бинарное выражение, в котором оператор – это арифметический оператор (`+`, `-`, `*`, `\`, `%`, `&`, `|`). Также проверяется, что это не случай конкатенации строк.
- `UnaryOp`. Применение унарного оператора `++` или `--` (префиксного или постфиксного).
- `ArithReassignment`. Бинарное выражение с оператором `+=`, `-=`, `*=`, `\=`, `%=`, `&=` или `|=`.
- `Comparison`. Сравнения `<`, `>`, `<=`, `>=`.

Правила представляют собой перечисление шаблонов, в которых не могут использоваться значения аннотированного `@Nullable` типа:

```
r {case ArithOp(rhs, lhs) =>
            require (notNull(rhs))
                    ("unboxing of nullable")
                    (POST_FACT)}
r {case ArithOp(rhs, lhs) =>
            require (notNull(lhs))
                    ("unboxing of nullable")
                    (POST_FACT)}
r {case UnaryOp(arg) =>
            require (notNull(arg))
                    ("unboxing of nullable")
                    (POST_FACT)}
```

```
r {case ArithReassignment(requiredType, expr)
            if requiredType.isPrimitive =>
            require (notNull(expr))
                    ("unboxing of nullable")
                    (POST_FACT)}
r {case Comparison(Var(v), expr)
            if v.`type`.isPrimitive =>
            require (notNull(expr))
                    ("unboxing of nullable")
                    (POST_FACT)}
r {case Comparison(expr, Var(v))
            if v.`type`.isPrimitive =>
            require (notNull(expr))
                    ("unboxing of nullable")
                    (POST_FACT)}
```

При приведении к примитивному типу проверяется, что приводимый объект ненулевой:

```
r {case TypeCast(expr, clazz)
            if clazz.`type`.isPrimitive =>
            require (notNull(expr))
                    ("unboxing of nullable")
                    (POST_FACT)}
```

Завершают список правил для анализа нулевых ссылок специальные правила для конструкций `throw-catch` и `synchronized`.

Пойманное исключение не может быть нулевым (по правилам языка Java):

```
f {case Catch(exception) =>
            fact (notNull(exception))}
```

Мы не можем запускать нулевое исключение (при попытке запустить `null` случится ошибка):

```
r {case Throw(exception) =>
            require (notNull(exception))
                    ("throwing nullable")}
```

Запрещается синхронизация по нулевым ссылкам:

```
r {case Synchronized(lock) =>
            require (notNull(lock))
                    ("locking nullable")}
```

Опять же, при попытке синхронизации по нулевой переменной происходит исключение.

4.1.2. Сравнение с Checker Framework

В разделе 5 далее будет описано сравнение производительности и объема кода, требуемого для реализации анализа. В данном разделе будет рассказано про функциональные отличия.

В пакете Checker Framework для анализа нулевых ссылок поддерживается множество дополнительных аннотаций (`@PolyNull`, `@LazyNonNull`, `@AssertNonNullAfter`, `@Pure` и др.). В данной работе под анализом нулевых ссылок понимается только семантика аннотаций `@NonNull` и `@Nullable`, соответственно, дальнейшее сравнение происходит только для этих аннотаций.

В Checker Framework реализовано множество функциональных тестов для анализа нулевых ссылок. Для проверки правильности анализа, реализованного на предлагаемом в работе языке, работоспособность была проверена на всех тестах Checker Framework для рассматриваемых аннотаций.

При этом результат выполнения тестов совпадает с результатом выполнения этих же тестов на Checker Framework с точностью до следующих отличий:

- в Checker Framework часто при аналогичных “нарушениях” генерируется только первое предупреждение об ошибке:

```
void method(@NonNull Object o) { ... }

void test(@Nullable Object arg) {
  method(arg);        //warning
  method(arg);
}
```

Но при выполнении приведенного выше фрагмента кода вероятна ситуация, в которой исключение случится только при втором вызове метода `method(arg)`. В этом случае получится, что о конкретном фрагменте кода, в котором случается исключение, не предупреждалось в результате анализа.

В данной работе в подобной ситуации генерируются два сообщения об ошибке. Повторное сообщение об ошибке не генерируется только в том случае, если точно произойдет ошибка (при значении переменной `null`), например, при разыменовании переменной.

- в Checker Framework не поддерживается правильная обработка неинициализированных `@NonNull` переменных (при наличии таких переменных должна генерировать ошибка, а она не генерируется):

```
class MyClass {
  @NonNull Object o;
  void method() {
```

```
            o.toString();
          }
        }
```

В приведенном выше примере при выполнении кода

```
MyClass c = new MyClass();
c.method();
```

случится исключение `NullPointerException`, а статический анализ Checker Framework этого не заметит. Последнее свидетельствует о том, что статический анализ нулевых ссылок в Checker Framework не является корректным (sound).

4.1.3. Корректность анализа

Утверждение. Анализ нулевых ссылок всегда завершает свою работу.

■ Как было показано в разделе 3.2.2, для доказательства того, что анализ завершает свою работу, достаточно показать, что в решётке абстрактных состояний для этого анализа нет бесконечных цепей. В данном случае это выполняется, так как решётка абстрактных состояний для анализа нулевых ссылок является конечной. Покажем это.

Аргументами отношения могут быть только поддеревья анализируемого абстрактного синтаксического дерева и элементы потока выполнения (переменные, методы), встречающиеся в анализируемом коде, а их конечное число. Про каждый элемент может быть сгенерировано два факта:

- `notNull(o)`,
- `not (notNull(o))`.

Поэтому, при анализе фрагмента кода может быть сгенерировано только конечное число фактов. Значит, множество возможных состояний (множеств фактов) конечно. □

Анализ нулевых ссылок является корректным (sound), то есть если в программе случается исключение `NullPointerException`, то результатом статического анализа будет предупреждение о возможной ошибке. Анализ нулевых ссылок не является полным (complete), то есть возможна ситуация, когда в результате статического анализа генерируется предупреждение об ошибке, а при выполнении программы исключения не происходит.

Неформальное объяснение корректности анализа заключается в следующем. Представим, что предупреждение об ошибке будет выдаваться при разыменовании любого объекта. Тогда, если в результате такого анализа не генерируется ни одной ошибки, то исключение `NullPointerException` гарантированно не случится (мы

заставили программиста добавить проверки при каждом разыменовании). Такой анализ является корректным, но для программиста такая ситуация не приемлема. Более тонкий анализ позволяет отдельно помечать объекты (аннотацией `@NonNull`), которые можно безопасно разыменовывать (в которых гарантированно не хранится значение `null`). Чтобы доказать, что этот более тонкий анализ является корректным, нужно доказать, что инвариант “В объекте, аннотированном `@NonNull`, не хранится значение `null`”, в процессе выполнения не может нарушиться. Для этого требуется анализ формальной спецификации семантики языка Java. Например, в работе [19] можно найти формальное доказательство аналогичного утверждения про корректность анализа нулевых ссылок (представленного в данной работе).

Утверждение о том, что анализ нулевых ссылок не является полным, достаточно очевидно. Например, при анализе следующего метода:

```
void method() {
    @Nullable String rrr = “rrr”;
    rrr.toString();                //warning
}
```

будет сгенерировано сообщение о возможной ошибке, однако, при выполнении исключения не случится.

4.2. Выявление неиспользованных значений переменных

Когда в программе переменной присваивается какое-то значение, предполагается, что это значение будет где-то использоваться (в противном случае такое присваивание является ненужным и его можно не писать). Если в коде встречается неиспользованное значение переменной, то с большой вероятностью в этом месте кроётся ошибка. Например, программист в том месте, в котором хотел использовать объявленную переменную, использовал другую переменную с похожим названием (например, предыдущее состояние того же объекта). Такая ошибка может быть незаметна сразу при выполнении программы. Или программист мог хотеть использовать объявленную переменную, а потом после нескольких написанных строк кода забыть об этом (в таком случае ошибки во время выполнения скорее всего не будет, однако лучше отслеживать такие ситуации). В любом случае такие ошибки трудно заметить в процессе написания кода, а потом может быть сложно найти. Обычно в среде разработки отслеживаются такие ситуации и еще во время написания кода выдаются предупреждения, которые легко заметить.

Данный статический анализ (нахождение неиспользованных значений переменных) на предлагаемом языке записывается очень просто.

4.2.1. Правила

Для осуществления данного статического анализа удобнее всего использовать обратный обход абстрактного синтаксического дерева. При обратном обходе все однотипные вершины (например, утверждения в методе) обходятся в обратном порядке.

Объявляется одноместное отношение “значение переменной было использовано”:

```
val used = UnaryRelation()
```

Если при изменении значения переменной её предыдущее значение не было использовано, будет сгенерировано предупреждение о возможной ошибке.

На предлагаемом языке достаточно записать всего три правила:

```
f {case VarRead(v) => fact (used(v)) }

r {case VarChange(v) => require (used(v))
         (“Unused variable”) (POST_NEG_FACT)}

r {case ArithReAssign(v) => require (used(v))
         (“Unused variable”) (POST_FACT)}
```

Первое правило для любого использования переменной (кроме изменения её значения), генерирует факт о том, что переменная была использована.

Второе правило применяется при объявлении переменной или присваивании переменной значения. В этом случае проверяется, что переменная была использована, если это не так, генерируется сообщение об ошибке. После этого благодаря параметру `POST_NEG_FACT` генерируется факт `not (used(v))`.

Третье правило применяется, если для изменения значения переменной используются арифметические операторы `+=`, `*=`, `-=`, `/=`, `%=`, `++` или `--`. В этой ситуации, во-первых, нужно проверить, что в дальнейшем значение переменной используется (так как происходит его изменение), во-вторых, сгенерировать факт о том, что предыдущее значение переменной было использовано.

Например, рассмотрим следующий фрагмент кода:

```
void method() {
  int i = 0;     //warning
  i = 1;
  i++;           //warning
}
```

В приведенном примере генерируются ошибки сразу о двух неиспользуемых значениях переменной:

- значение `0`, присвоенное переменной `i` при объявлении, далее не используется, так как переменной `i` присваивается другое значение;
- значение, полученное после выполнения команды `i++`, также в дальнейшем не используется.

Процесс статического анализа происходит следующим образом:

- Анализируется выражение `i++`. Соответствующее ему абстрактное синтаксическое дерево удовлетворяет шаблону `ArithReAssign`, однако в текущем состоянии нет факта об использовании переменной `i`, поэтому генерируется сообщение об ошибке. Также генерируется факт `used(i)`.
- Анализируется выражение `i = 1`. Оно удовлетворяет шаблону `VarChange`, для переменной `i` есть факт `used(i)`, потенциальных ошибок нет. После этого генерируется факт `not (used(i))`.
- Анализируется объявление переменной `int i = 0`. Факта об использовании переменной нет, поэтому генерируется сообщение об ошибке.

4.2.2. Корректность анализа

Утверждение. Анализ для выявления неиспользованных значений переменных всегда заканчивает свою работу.

Доказательство аналогично доказательству подобного утверждения для анализа нулевых ссылок (см. раздел 4.1.3).

4.3. Выявление бессмысленных сравнений и присваиваний

Другая ситуация, которая с большой вероятностью свидетельствует об ошибке: наличие бессмысленного сравнения. Условие, которое заведомо истинно или ложно, можно не писать вовсе. Если оно встречается, скорее всего, подразумевалась какая-то другая логика или что-то забыто. Например, забыв изменить значение счётчика, можно получить бесконечный цикл:

```
int i = 0;
while (i < 20) {      //warning
   // в теле цикла не меняется значение переменной i
}
```

Конечно, такая ошибка при первом запуске программы будет обнаружена, но могут быть менее заметные подобные ошибки.

Также бессмысленным действием является присваивание переменной значения, которое в ней уже содержится.

Для предотвращения подобных ошибок применяется статический анализ в процессе написания кода, и программисту сразу указывается на “странный” код.

4.3.1. Правила

Для описания семантики данного анализа объявляется бинарное отношение, являющееся функцией:

```
val varValue = Function[VariableValue]()
```

Вторым аргументом отношения является значение типа `VariableValue`. Класс `VariableValue` – это абстрактный класс, у которого есть два наследника:

Рис 4.1. Возможные значения типа VariableValue

При этом у класса Reference() есть внутренний счётчик, который отслеживает заведение новой ссылки. То есть при каждом вызове Reference() заводится объект “ссылка” с текущим номером, и количество ссылок увеличивается.

Объединение и пересечение объектов определено следующим образом:

- при объединении или пересечении одинаковых объектов (ссылок с равными номером или объектов с равными примитивными значениями) результатом является этот объект;
- при объединении разных объектов (ссылки и примитивного значения, разных ссылок, разных примитивных значений) результатом является специальное значение `Undefined` (что в отношении эквивалентно отсутствию факта про элемент);
- при пересечении разных объектов результатом является специальное значение `Bottom`.

Введенные типы и специальные значения `Undefined` и `Bottom` образуют бесконечную решетку (см. Рис 4.2).

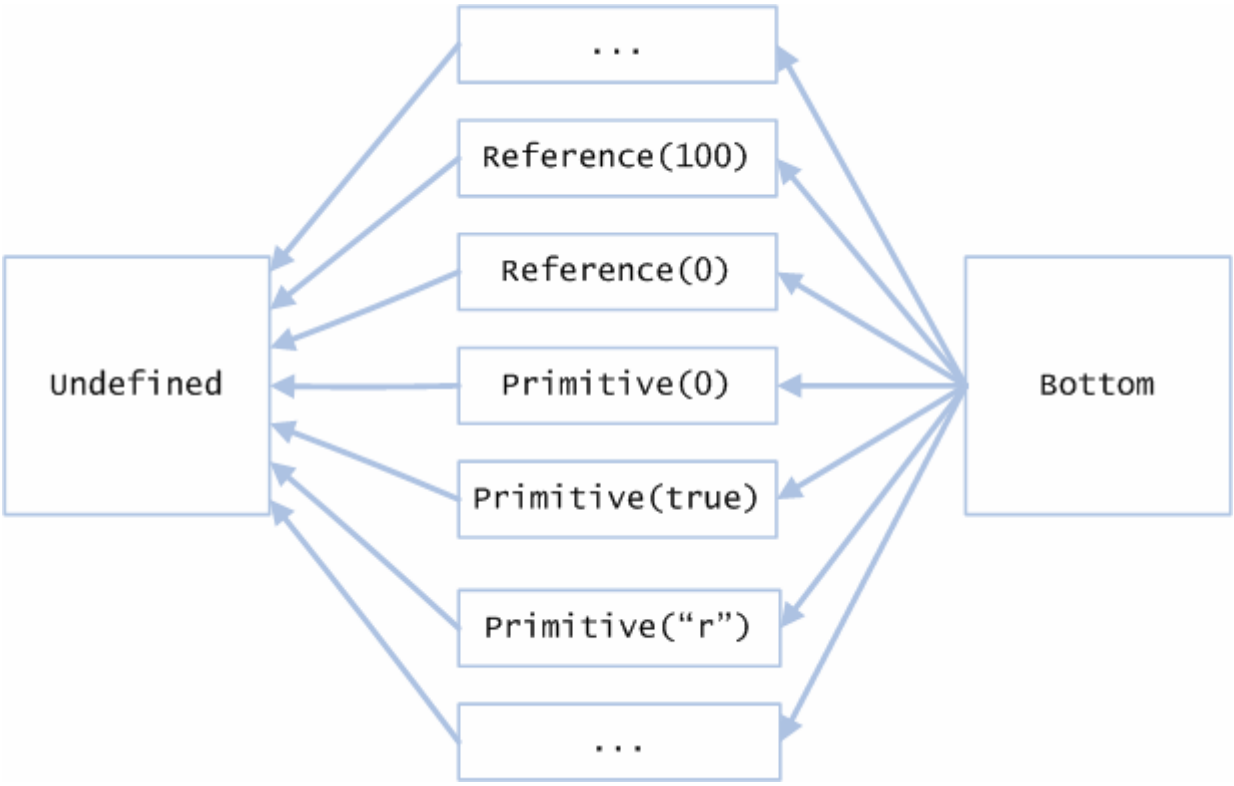


Рис 4.2. Бесконечная решетка, которую образуют введенные типы и специальные значения

Далее описываются правила, задающие семантику анализа.

При создании нового объекта с помощью конструкции `new` для переменной, в которую присваивается ссылка на данный объект, сохраняется факт, что ее значение – это новая ссылка:

```
f { case VarAssignment(x, NewClass(_)) =>
                        fact (varValue(x, Reference())) }
f { case VarAssignment(x, NewArray(_)) =>
                        fact (varValue(x, Reference())) }
```

При присваивании переменной примитивного значения эта информация запоминается. При этом если происходит не объявление переменной, а присваивание уже существующей переменной нового значения, то проверяется, что в ней не записано такое же значение.

```
f { case VarDecl(x, Literal(obj)) =>
                        fact (varValue(x, Primitive(obj))) }
r { case Assignment(Var(x), Literal(obj)) =>
                        check (varValue(x), Primitive(obj))
                              ("Senseless assignment!")
                              (POST_FACT)}
```

В последнем правиле используется ключевое слово `check`. Оно означает, что генерируется предупреждение об ошибке, если оказалось, что отношение выполняется.

В следующих правилах используется синтаксис `=`, `==`, `!=` для сравнения значений функций (отношение varValue было объявлено как функция).

Если переменной присваивается значение другой переменной, мы запоминаем этот факт:

```
f { case VarDecl(x, Var(y)) =>
                        fact (varValue(x) `=` varValue(y)) }
r { case Assignment(Var(x), Var(y)) =>
                        check (varValue(x `=` varValue(y)))
                              ("Senseless assignment!")
                              (POST_FACT)}
```

При этом значением функции varValue(x) становится значение функции varValue(y). То есть если в абстрактном состоянии был факт про отношение varValue(y, Primitive(42)), то после выполнения правила в состояние добавится факт varValue(x, Primitive(42)).

Если переменной присваивается значение, отличное от вышеперечисленных (не ссылка на новый создаваемый объект, не примитивное значение, не значение другой переменной), то мы “забываем” всю информацию о значении переменной:

```
f { case VarAssignment(x, CompoundExpression(_)) =>
                        fact (varValue(x, Undefined)) }
```

Если значение переменной изменяется с помощью арифметического оператора, информация о значении переменной забывается:

```
f { case ArithReAssign(x) =>
                        fact (varValue(x, Undefined)) }
```

Далее перечисляются правила, которые позволяют отслеживать ситуации, когда производится заведомо истинное или заведомо ложное сравнение.

```
r { case Equality(Var(x), Var(y)) =>
                        check (varValue(x) `==` value(y))
                              ("Senseless comparison (always true)!") }
r { case Inequality(Var(x), Var(y)) =>
                        check (varValue(x) `!=` varValue(y))
                              ("Senseless comparison (always true)!") }
r { case Equality(Var(x), Literal(obj)) =>
                        check (varValue(x) `==` Primitive(obj))
                              ("Senseless comparison (always true)!") }
r { case Inequality(Var(x), Literal(obj)) =>
                        check (varValue(x) `!=` Primitive(obj))
                              ("Senseless comparison (always true)!") }
```

```
r { case Equality(Var(x), Var(y)) =>
                    check (varValue(x) `!=` varValue(y))
                          ("Senseless comparison (always false)!") }
r { case Inequality(Var(x), Var(y)) =>
                    check (varValue(x) `==` varValue(y))
                          ("Senseless comparison (always false)!") }
r { case Equality(Var(x), Literal(obj)) =>
                    check (varValue(x) `!=` Primitive(obj))
                          ("Senseless comparison (always false)!") }
r { case Inequality(Var(x), Literal(obj)) =>
                    check (varValue(x) `==` Primitive(obj))
                          ("Senseless comparison (always false)!") }
```

4.3.2. Корректность анализа

Утверждение. Анализ для выявления бессмысленных сравнений и присваиваний всегда завершает свою работу.

■ Аналогично доказательству для анализа нулевых ссылок (см. раздел 4.1.3) покажем, что решётка абстрактных состояний является конечной.

Теперь у отношения два аргумента, но опять множество первых возможных аргументов (элементов анализа кода) конечно. При этом множество вторых возможных аргументов также конечно. В самом деле, вторым аргументом отношения может быть одно из следующих значений:

- Reference,
- Primitive,
- Undefined,
- Bottom.

Для анализируемого фрагмента кода может быть создано конечное количество объектов типа `Reference` (по одному для каждого создания объекта с помощью оператора `new`) и конечное количество объектов типа `Primitive` (по одному для каждой числовой константы или строки, встречающейся в данном фрагменте кода).

Заметим, что при анализе циклов не будут генерироваться новые и новые ссылки (если в теле цикла встречается оператор `new`), так как при объединении разных ссылок, происходящем после каждой итерации при анализе цикла, мы сразу же получим значение `Undefined`.

Мы получили, что число фактов, которое может быть сгенерировано для данного фрагмента кода конечно (факт – это бинарное отношение `varValue` с первым аргументом из конечного множества элементов и со вторым аргументом из конечного множества

значений). Поэтому, множество возможных состояний (множеств фактов) также конечно. □

4.4. Выявление непроверенных приведений к типу

В языке Java при некорректном приведении к типу случается исключение `ClassCastException`:

```
Object o = “str”;
Integer i = (Integer)o; //ClassCastException
```

Для предотвращения подобных ошибок стоит перед приведением к типу проверить допустимость этого приведения:

```
Object o = “str”;
if (o instanceOf Integer) {
  Integer i = (Integer)o;
}
```

С помощью предлагаемого языка реализован статический анализ, который при приведении к типу требует предварительных проверок допустимости такого приведения. Если предварительных проверок не было, генерируется сообщение о возможной ошибке.

4.4.1. Правила

Объявляется бинарное отношение “быть экземпляром класса”, которое является функцией:

```
val instanceOf = Function[Type]()
```

Значениями данной функции будут элементы типа `Type`.

Объединение двух типов – это наиболее общий тип (часто это может быть `Object`), пересечение – специальное значение `Bottom`.

Определяется два правила. Если в условии встречается проверка на принадлежность к типу, результат этой проверки запоминается:

```
c { case InstanceOf(Var(v), ClassRef(clazz)) =>
              fact (instanceOf(v, Type(clazz.`type`))) }
```

При попытке приведения выражения к типу, требуется наличие факта о предварительной проверки возможности такого приведения:

```
r { case TypeCast(expr, ClassRef(clazz)) =>
              require (instanceOf(expr, Type(clazz.`type`)))
```

```
("Unchecked cast!")}
```

4.4.2. Корректность анализа

Утверждение. Анализ для выявления непроверенных приведений к типу всегда завершает свою работу.

■ Доказательство аналогично предыдущему случаю (для анализа выявления бессмысленных сравнений и присваиваний). □

5. Сравнение с аналогами на примере анализа нулевых ссылок

В данном разделе приведены результаты сравнительного анализа реализаций анализа нулевых ссылок в пакетах Checker Framework, JavaCOP и на предлагаемом языке. В разделе 5.1 приведены результаты сравнения объёма необходимого для описания семантики кода. В разделе 5.2 приведены результаты сравнения времени работы.

5.1. Объем кода

Часто декларативное описание функциональности выгоднее императивного с точки зрения объёма требуемого для написания (и тестирования) кода. Декларативный язык, предлагаемый в данной работе, не является исключением.

В Таблица 1 представлено сравнение количества строк кода, написанного с помощью разных средств описания семантики аннотаций на типах для реализации анализа нулевых ссылок. При подсчёте не учитывались комментарии и пустые строки.

Таблица 1. Сравнение объема кода, написанного для реализации анализа нулевых ссылок с помощью различных средств описания семантики аннотаций на типах

Средство описания семантики анализа	Количество строк кода
Checker Framework	> 1000 (java)
JavaCOP	111 (java)
	97 (jcop)
Предлагаемый язык декларативных правил	36 (scala)

В пакете Checker Framework для анализа нулевых ссылок реализована поддержка достаточно большого количества дополнительных аннотаций (@PolyNull,

`@LazyNonNull, @AssertNonNullAfter, @Pure` и др.). При подсчёте объема необходимого для реализации анализа нулевых ссылок кода интересовал только код, написанный для поддержки `@Nullable` и `@NonNull` аннотаций. Чтобы не производить разделение кода на код, описывающий рассматриваемые аннотации и код, описывающий все остальные, приводится достаточная для сравнения нижняя оценка.

Для пакета JavaCOP в таблице приведены следующие значения:

- количество строк кода на Java, на котором описывается семантика условий;
- количество строк кода на декларативном языке (jcop), реализованном в пакете JavaCOP, на котором описываются основные правила.

5.2. Время работы

Было проделано сравнение быстродействия реализаций анализа нулевых ссылок в пакете Checker Framework и на предлагаемом языке. Был произведен запуск решений на множестве тестовых примеров, приведенных в Checker Framework, на которых тестировалась функциональность предлагаемого решения. Было измерено точное время работы расширения компилятора, отвечающего за проведение анализа (а не всего процесса компиляции). Приведенные результаты представлены в Таблица 2.

Таблица 2. Сравнение быстродействия реализаций анализа нулевых ссылок в предлагаемом решении и пакете Checker Framework

	Предлагамое решение	Checker Framework	Относительное замедление
Flow.java	1092	204	5,35
Expressions.java	509	64	7,95
Casts.java	94	20	4,70
AnnotatedGenerics.java	827	71	11,65
FlowAssignment.java	42	2	21,00
FlowJavaCop.java	864	54	16,00
Boxing.java	242	45	5,38
DefaultLoops.java	131	10	13,10
ArrayArgs.java	45	5	9,00
ArrayRefs.java	59	9	6,56
DefaultFlow.java	32	3	10,67
Exceptions.java	71	7	10,14
JavaCopExplosion.java	293	62	4,73
LogicOperations.java	159	12	13,25
Marino.java	152	18	8,44
AnnotatedTypeParams.java	44	12	3,67

MyException.java	36	8	4,50
NullnessBound.java	29	9	3,22
SuperCall.java	7	2	3,50
Synchronization.java	13	6	2,17
Varargs.java	84	18	4,67
WildcardSuper.java	168	16	10,50
Общее время работы	4993	657	7,60

Измерения производились на компьютере со следующими характеристиками:

- процессор Intel Core2 Duo P9300 с частотой 2.27 GHz;
- оперативная память 2 Гб.

Из таблицы видно, что на небольших примерах замедление предлагаемого решения относительно Checker Framework происходит в среднем в 8 раз.

В результате анализа выяснилось, что 60% времени работы предлагаемого решения занимает процедура сравнения с образцом. В реализованном решении при применении правил происходит попытка применить все написанные правила к каждому поддереву абстрактного синтаксического дерева.

Способ улучшения быстродействия реализованной системы описан далее. Обход абстрактного синтаксического дерева осуществляется в соответствующем `Visitor` классе, где для вершины каждого типа есть свой `visit` метод:

```
visitFieldAccess(FieldAccessTree tree, …)
visitForLoop(ForLoopTree tree, …)
visitIf(IfTree tree, …)
visitIdentifier(IdentifierTree tree, …)
```

В реализованном решении при обходе вершины её тип “забывается” и происходит попытка применить все возможные правила к ней. Хотя на самом деле достаточно применять только правила, точно подходящие к этой вершине. Для реализации этого более быстрого подхода необходимо предварительно скомпилировать правила, разобрав их по типам вершин, к которым их можно применять. То есть сейчас классификация правил по типам вершин, к которым правила применимы, происходит в процессе выполнения для каждой вершины, а на самом деле классификация может быть выполнена заранее.

Заключение

В настоящей работе предложен декларативный язык правил для задания семантики аннотаций на типах. Язык реализован как встроенный доменно-специфичный язык на языке Scala. Реализована поддержка данного языка как расширение над Checker Framework.

Для решения поставленной задачи произведен сравнительный анализ доступных пакетов для задания семантики аннотаций на типах Checker Framework и JavaCOP, выделены основные достоинства и недостатки каждого пакета.

С помощью разработанного языка реализованы следующие примеры статического анализа кода:

- анализ нулевых ссылок;
- выявление неиспользованных значений переменных;
- выявление бессмысленных сравнений и присваиваний;
- выявление непроверенных приведений к типу.

Произведен сравнительный анализ задания семантики анализа нулевых ссылок в Checker Framework, в JavaCOP и на предлагаемом языке. Предложенный декларативный язык позволил сильно сократить объем кода, необходимого для описания семантики. Сравнение быстродействия выявило замедление относительно реализации анализа в Checker Framework, однако предложены методы повышения производительности решения.

Литература

1. Chess B., West Y., *Secure Programming with Static Analysis.* Addison-Wesley, 2007.

2. Surhone L., Timpledon M., Marseken S. *Static Code Analysis*. Betascript Publishing, 2010

3. Müller-Olm M. *Variations on Constants: Flow Analysis of Sequential and Parallel Programs.* Springer, 2006.

4. Gosling J., Joy B., Steele G., Bracha G. *The Java Language Specification.* Addison Wesley, Boston, MA, third edition, 2005.

5. JDK 7 features, http://openjdk.java.net/projects/jdk7/features/

6. Ernst M.D., *Type Annotations Specification (JSR 308)*, http://www.jcp.org/en/jsr/detail?id=308.

7. *Type Annotations (JSR 308) and the Checker Framework*, http://types.cs.washington.edu/jsr308/

8. Papi M., Ali M., Correa T.L., Perkins J., Ernst M.D., *Practical Pluggable Types for Java*. ISSTA, Static Analysis, July 20–24, 2008

9. Dietl W., Dietzel S., Ernst M.D., Muslu D., Schiller T.W., *Building and Using Pluggable Type-Checkers*. ICSE, Software Engineering at Large, May 21–28, 2011.

10. Baral C., *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003

11. Lloyd J.W., *Practical Advantages of Declarative Programming*. Department of Computer Science, University of Bristol.

12. Sebesta R. W., *Concepts of Programming Languages.* Addison-Wesley Publishing Company, 1996.

13. Markstrum S., Marino D., Esquivel M., Millstein T., Andreae C., Noble J., *JavaCOP: Declarative pluggable types for Java.* ACM TOPLAS, 32(2):1–37, Jan. 2010.

14. Aho A., Sethi R., Ullman J. *Compilers. Principles, Techniques and Tools*. Addisson-Wesley, 1985

15. Ekman T., Hedin G. *Pluggable checking and inferencing of non-null types for Java*. J. Object Tech., 6(9):455–475, Oct. 2007.

16. Fahndrich M., Leino K. *Declaring and checking non-null types in an object oriented language*. In Proc. OOPSLA, pages 302–312. ACM Press, 2003.

17. Chalin P, James P. *Non-null references by default in Java: Alleviating the nullity annotation burden*. In Proc. ECOOP, pages 227–247. Springer, 2007.

18. Hovemeyer D., Spacco J., Pugh W. *Evaluating and tuning a static analysis to find null pointer bugs*. In Proc. PASTE, pages 13–19. ACM Press, 2005.

19. Male C., Pearce D., Potanin A., Dymnikov C. *Java Bytecode Verification for @NonNull Types*. In Proc. ISSTA, ACM Press, 2008

20. Odersky M., Spoon L., Venners B. *Programming in Scala, Second Edition*. Artima, 2010

21. Wampler D., Payne A. *Programming Scala. Scalability = Functional Programming + Objects*. O'Reilly Media, 2009

22. Loverdos C., Syropoulos A. *Steps in Scala. An Introduction to Object-Functional Programming*. Cambridge University Press, 2010

23. Geller F., Hirschfeld R., Bracha G. *Pattern Matching for an Object-oriented and Dynamically Typed Programming Language*. Hasso-Plattner-Institute, HPI Technical Reports, vol. 36, 2010.

24. Кирпичёв Е. *Элементы функциональных языков*. Практика функционального программирования, выпуск 3, 2009

25. Cousot P., Cousot R. *Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints.* POPL '77 Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, 1977

26. Cousot P. *Semantic foundations of program analysis.* In Muchnick S., Jones N., editors, Program Flow Analysis: Theory and Applications, Ch. 10, pages 303—342, Prentice-Hall, 1981.

27. Cousot P., *Abstract interpretation*. MIT course 16.399, http://web.mit.edu/16.399/www/, February-May, 2005.

28. Леошкевич И. *Анализ зависимостей в исполняемом коде*. Сборник научных трудов. Т.5:Информационно-телекоммуникационные системы. Проблемы информационной безопасности. МИФИ, 2010.

29. Davey B., Priestley, H. *Introduction to Lattices and Order*. Cambridge University Press, 2002

30. Birkhoff G. *Lattice Theory, 3rd ed*. Vol. 25 of AMS Colloquium Publications, American Mathematical Society, 1967.

31. Dilworth R., Crawley P. *Algebraic Theory of Lattices.* Prentice-Hall, 1973.

32. Granas A., Dugundji J. *Fixed Point Theory*. Springer-Verlag, New York, 2003.