Индексирование библиотек Java для вывода контрактов

Илья Ключников

Аннотация

В статье рассматривается подход, обеспечивающий производительность и модульность вывода контрактов для библиотек Java. Методы библиотеки независимо друг от друга индексируются в один проход. В построенных индексах в компактной форме содержатся уравнения, необходимые для вывода контрактов. При выводе контрактов для набора зависимых библиотек, по индексам строится и решается система уравнений. Построение и решение системы не требовательно к ресурсам, на порядок быстрее, чем индексация библиотек, и не является узким местом.

1 Введение

Данная статья – вторая в цикле учебных статей, рассказывающих, как устроен вывод контрактов для Java библиотек в IDEA 14.

Статьи разбирают по шагам два самодостаточных учебных проекта: *Kanva-micro* и *Faba*.

- В проекте *Kanva-micro* [3] основное внимание уделяется сути метода – суперкомпиляции над абстрактными значениями и выводу аннотаций по графу конфигураций.
- Проект *Faba* [2] решает проблемы производительности и модульности, которые обходятся в проекте *Kanva-micro* стороной.

Проект *Kanva-micro* был разобран в первой статье цикла [10]. Данная статья посвящена одному из аспектов реализации проекта *Faba* – индексированию библиотек для вывода контрактов. Другие важные аспекты – минимизация числа состояний в графе конфигураций и устранение избыточных шагов при анализе – будут рассмотрены в будущих статьях цикла.

1.1 Контрактное программирование в IntelliJ IDEA

IntelliJ IDEA – среда разработки для Java, выпускаемая компанией JetBrains. IDEA поддерживает контрактное программирование и позволяет определять контракты через аннотирование кода дополнительной информацией, а инструменты анализа качества кода предупреждают программиста о нарушениях контрактов.

Аннотации `@NotNull` и `@Nullable` используются при анализе кода на предмет разыменовывания нулевых ссылок (для предотвращения `NullPointerException` во время исполнения кода).

Аннотация `@Nullable` служит явным напоминанием о том, что соответствующее значение (переменная, результат вызова метода, параметра метода) может быть `null`. Аннотация `@NotNull`, в свою очередь, используются для определения контракта, что

- метод не возвращает `null`,
- переменная (локальная переменная, поле, параметр метода) не предназначена нулевых ссылок.

Среди прочего, анализ кода предупреждает пользователя о конфликтах между `@Nullable` и `@NotNull` значениями. Например, когда `@Nullable` значение передается в качестве аргумента в `@NotNull` параметр.

IDEA также поддерживает аннотации `@Contract`, которые позволяют указать зависимости между параметрами и результатом метода. Например:

```
@Contract("null->true")
public static boolean isEmpty(Object[] array) {
  return array == null || array.length == 0;
}

@Contract("!null->!null")
public static String defaultString(String str, String defaultStr) {
  return str == null ? defaultStr : str;
}
```

`@Contract` аннотации оперируют следующими значениями (называемые далее для краткости *контрактными значениями*): `_` (любое значение), `false`, `true`, `null`, `!null`, `fail` (генерация исключения). `@Contract` аннотации также учитываются инструментами анализа кода.

Аннотации `@NotNull`, `@Nullable` и `@Contract` указываются программистом в исходном коде. Для аннотирования используемых сторонних библиотек можно использовать механизм внешних аннотаций – аннотации указываются во внешнем xml-файле специального формата.

1.2 Вывод контрактов в IDEA 14

В выходящем в конце этого года релизе IDEA 14 методы из библиотек Java, используемых в проекте, автоматически размечаются аннотациями `@NotNull` и `@Contract`.

Параметр метода аннотируется как `@NotNull`, если в любых обстоятельствах при передаче `null` в данный параметр метод не может завершиться нормально (без исключения). Метод аннотируется `@NotNull`, если значение, возвращаемое данным методом, не может быть `null`.

Аннотация `@Contract("in->out")` добавляется к методу, если по контрактному значению параметра `in` удается однозначно установить контрактное значение результата `out`.

Технически, используемые в проекте библиотеки заранее индексируются и аннотируются в фоновом режиме.

1.3 Проект *Faba*

Реализация автоматического аннотирования в IDEA 14 интегрирована с общей инфраструктурой IDEA и включает в себя множество микрооптимизаций, чтобы аннотирование могло выполняться быстро и не было требовательным по памяти на больших проектах.

Соответственно, неподготовленному читателю, разобраться в этой реализации может быть достаточно сложно.

Проект *Faba* представляет из себя реализацию автоматического аннотирования, не привязанную к IDEA, и является инструментом, который можно запустить из командной строки. Из-за минимализма проект *Faba* также удобен для экспериментов. В данной статье подробно разбирается вывод `@NotNull` аннотаций для параметров. Вывод остальных аннотаций организован схожим образом.

1.4 Организация статьи

Дальнейший текст организован следующим образом. В разделе 2 кратко описываются основы вывода `@NotNull` аннотаций через суперкомпиляцию на абстрактных значениях и анализ графа конфигураций и перечисляются технические проблемы, возникающие при наивной реализации метода. В разделе 3 разбирается технический прием для обеспечения модульности анализа – составление по графу конфигураций уравнений на решетках и их последующее решение, приводятся примеры и экспериментальные данные. В разделе 4 подводится итог и рассматриваются направления дальнейших работ.

2 Основная идея алгоритма

В данном разделе кратко повторяются основные моменты вывода `@NotNull` аннотаций на параметрах, подробно описанного в статье [10]. Далее предполагается, что читатель знаком с основами суперкомпиляции [17, 11].

Основная идея заключается в следующем. Предполагается, что в интересующий нас параметр некоторого метода передан `null`. Рассматриваются все возможные пути исполнения этого метода. Если на каждом из возможных путей исполнения возникает ошибка, причиной которой является переданная нулевая ссылка, то параметр аннотируется как `@NotNull`.

2.1 Построение графа конфигураций

Построение всех возможных путей основано на методах суперкомпиляции – строится граф конфигураций, в котором содержатся все пути исполнения данного метода. Для целей аннотирования достаточно все возможные значения, с которыми работает виртуальная машина Java, разделить на два класса значений – `ParamValue` (иногда обозначаемое далее для краткости символом •) и `BasicValue` (иногда обозначаемое далее для краткости символом ∘). `ParamValue` – значение, переданное в параметр, `BasicValue` – любое значение. Если рассматривать `ParamValue` и `BasicValue` как множества, то `ParamValue` содержится в `BasicValue`: • ⊆ ∘.

Итак, при построении графа конфигураций для метода, все *значения* переменных, с которыми работает данный метод обобщаются до • и ∘. Далее, исполнение всех инструкций байткода Java обобщается до исполнения над абстрактными значениями • и ∘ – это делается с помощью достаточно простого интерпретатора. Граф конфигураций по инструкциям байткода рассматриваемого метода, вложенные вызовы не разворачиваются (интрапроцедурный анализ). Конфигурация представляется как пара `insnIndex, frame`, где `insnIndex` – номер текущей инструкции внутри метода (все инструкции внутри метода заранее нумеруются), а `frame` – обобщенное состояние текущего фрейма, представленное в виде списка из • и ∘. Конфигурация c является подконфигурацией (частным случаем) конфигурации c', если:

- индексы инструкций (`insnIndex`) совпадают и
- соответствующие элементы фреймов относятся как $v_i \subseteq v'_i$.

Начальная конфигурация конструируется следующим образом: элемент фрейма, соответствующий аннотируемому параметра, инициализируется как •, все остальные элементы фрейма – как ∘. Граф конфигураций строится классическим образом. Если встречается развилка, на которой сравнивается `null` и •, то рассматривается только ветка, соответствующая нулевому значению параметра. Поддерево, следующее за этим переходом, называется *`null`-информированное поддерево*. На графах конфигурациях узлы `null`-информированного поддерева обозначаются двойными контуром.

Свертка происходит, если обнаруживается, что текущая конфигурация является частным случаем некоторой предшествующей ей конфигурации. Обобщение не требуется, так как количество всех возможных конфигураций конечно.

При построении переходов между конфигурациями отслеживаются конфигураций, исполнение инструкций которых приводит к разыменовыванию аннотируемого параметра. Условия разыменовывания можно неформально описать так:

1. У параметра вызывается метода экземпляра или происходит чтение или запись полей параметра. В этом случае конфигурация помечается специальной меткой ⊖.

2. Текущий параметр • передается как аргумент в (другой) параметр, который в свою очередь не принимает нулевые ссылки (проаннотированный как `@NotNull` параметр). Такая конфигурация также помечается как ⊖.

Листья дерева, не являющиеся частью свертки, называются *терминальные* листья.

Схематичный пример графа конфигураций приведен на Рис. 1. В узле c_2 обнаружено разыменовывание параметра. Поддерево, состоящее из единственного узла c_6 – `null`-информированное поддерево. Узлы c_4 и c_6 – терминальные листья.

2.2 Вывод аннотаций по графу конфигураций

Рассматриваются терминальные листья графа конфигураций. Параметр аннотируется как `@NotNull`, если для всех терминальных листьев выполняется любое из следующих условий:

- На пути от корневого узла до данного терминального листа есть узел с меткой ⊖.

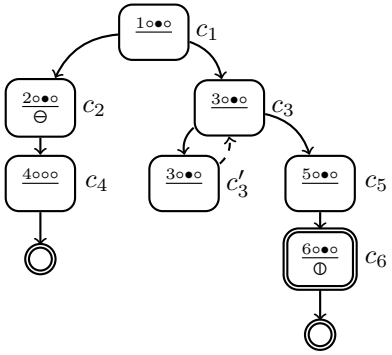

Рис. 1: Пример графа конфигураций

- На пути от корневого узла до данного терминального листа внутри `null`-информированого дерева инициализируется исключение.

Технически вычисление этих условий сводилось к тому, что все узлы графа маркировались специальными метками `RETURN`, `NPE`, `ERROR`, `CYCLE`, начиная с листьев, и затем вычислялась метка корневого узла. Если метка корневого узла оказалась `NPE`, то параметр аннотировался как `@NotNull`.

Пусть в узле c_6 на Рис. 1 инициируется исключение. Соответствующий параметр аннотируется как `@NotNull`, так как на пути от корня до терминального листа c_4 есть узел c_2 с меткой $\ominus$, и на пути от корня до терминального листа c_6 в узле c_6, принадлежащему `null`-информированному поддереву, инициируется исключение.

2.3 Проблемы наивной реализации

Однако, в *Kanva-micro* наивная реализации вывода аннотаций на основе описанного метода обладает следующими недостатками.

2.3.1 Многопроходность и монолитность

В общем случае аннотирование одного параметра может зависеть от результата аннотирования другого параметра.

Рассмотрим простейший пример на Рис. 2. В этом примере аннотирование первого параметра `A.foo` зависит от аннотирования первого параметра `B.bar`. *Kanva-micro* вначале строит граф зависимостей между методами, выявляет в нем строго связанные компоненты и затем обрабатывает компоненты в обратном топологическом порядке. При об-

```
class A {
  static void foo(Object o1, Object o2) {
    B.bar(o1, o2, false);
  }
}

class B {
  static void bar(Object o, Object o2, boolean b) {
    ...
  }
}
```

Рис. 2: Пример зависимых методов

работке компоненты все параметры анализируются итеративно с учетом зависимостей: если на данной итерации параметр был проаннотирован как `@NotNull`, то зависимые от него параметры аннотируются заново.

Таким образом, каждый метод должен быть проанализирован как минимум два раза: первый раз при анализе зависимостей, а затем (возможно, несколько раз) при выводе `@NotNull` аннотаций.

Издержки такого подхода ощутимы при анализе больших библиотек или при анализе большого количества зависимых библиотек. С одной стороны, если инструкции байткода метода каждый раз считываются с диска, это заметно сказывается на производительности. С другой стороны, если если минимизировать чтение байткода и кэшировать байткод в памяти, реализация становится требовательна к памяти.

Индексация библиотек, описываемая в статье, решает эти проблемы.

2.3.2 Избыточность графа конфигураций

В общем случае размер графа конфигураций, построенного по инструкциям метода классическим способом, экспоненциально зависит от количества ветвлений в методе. Здесь проявляется императивная природа программирования на Java. В коде на Java часто встречаются последовательные (не вложенных в друг друга) `if`-выражения. Рассмотрим схематичный пример на Рис. 3. В теле метода параметр `arr` используется только в последней строке, и только эта строка важна для вывода `@NotNull` аннотации параметра `arr`. Однако, построенный граф конфигураций будет иметь вид, приведенный на Рис. 4 (некоторые детали опущены). Видно, что такой граф избыточен – в нем много повторяющихся частей. В случае сложных и длинных методов с большим количеством ветвлений, такая избыточность существенно влияет на размер

```
static int getValue(int[][] arr, int i1, int i2, int i3) {
  int x;
  if (i1 == i2) {
    x = ...;
  } else if (i1 > i2) {
    x = ...;
  } else {
    x = ...;
  }
  int y;
  if (i2 == i3) {
    y = ...;
  } else if (i2 > i3) {
    y = ...;
  } else {
    y = ...;
  }
  return arr[x][y];
}
```

Рис. 3: Пример источника избыточности графа конфигураций

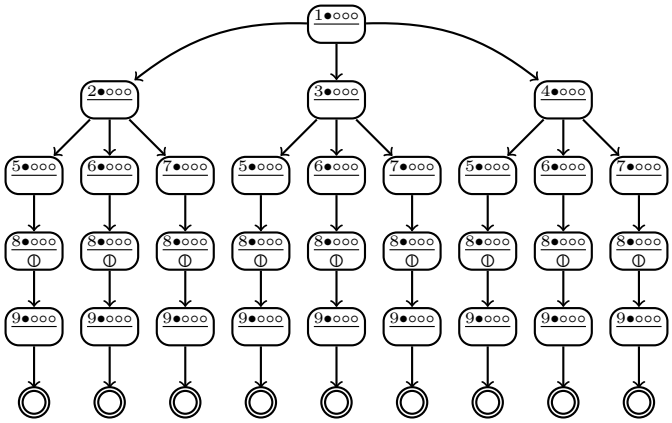

Рис. 4: Избыточный граф конфигураций

графа конфигураций.

Данная проблема решена в проекте *Faba*, подробное решение будет описано в будущих статьях.

3 Индексирование

В *Kanva-micro* анализ зависимостей между методами и анализ нулевых ссылок (через анализ графа конфигураций) были разделены. Анализ

зависимостей учитывал только зависимость между методами и не учитывал деталей анализа нулевых ссылок. Такой анализ зависимостей в контексте решаемой задачи был избыточным – если метод `g` использовался внутри метода `f`, то метод `f` объявлялся зависимым от метода `g` – в общем случае результат аннотирования `g` влияет на аннотирование `f`.

В следующем примере *Kanva-micro* объявит метод `f` зависимым от `g`, однако аннотирование `o1` не зависит от результатов аннотирования параметров метода `g`.

```
static void f(Object o1, Object o2) {
  g(o2);
  h(o1);
}
```

Если же делать точный анализ зависимостей параметров в контексте решаемой задачи, то такой анализ становится очень похожим на анализ нулевых ссылок.

Основная идея индексирования заключается в составлении в компактном виде уравнений, достаточных для вывода контрактов. При индексировании совмещается построение графа конфигураций, анализ графа конфигураций и анализ зависимостей. Избыточность анализа зависимостей устраняется, так как учитываются только те зависимости, которые действительно влияют на результат аннотирования.

Архитектурно *Faba* состоит из двух компонент

- Решатель уравнений над решетками.
- Фреймворк для конструирования уравнений над решетками по графу конфигураций.

Как оказалось, задача вывода всех аннотаций `@NotNull` и `@Contract` сводима к системе уравнений над решетками.

3.1 Устройство уравнений

На вход решатель *Faba* получает систему уравнений $\overline{eq_i}$ над ограниченными полными решетками (complete bounded lattices) и выдает решение $\overline{sol_i}$. Синтаксис уравнений, с которыми работает решатель, представлен на Рис. 5. Решатель может работать с любой ограниченной полной решеткой. Решетка, которая используется для вывода аннотаций `@NotNull` и `@Contract` представлена на Рис. 6. Множества `false`, `true`, $\bullet$, $\overline{\bullet}$ являются попарно непересекающими. То есть для любых элементов $l_i \neq l_j$ из `false`, `true`, $\bullet$, $\overline{\bullet}$ верно:

$$l_i \sqcup l_j = \top \qquad\qquad l_i \sqcap l_j = \bot$$

eq	$::=$	$v_i = rhs$	уравнение
rhs	$::=$	sop	правая часть уравнения
sop	$::=$	$\bigsqcup \overline{prod}$	сумма произведений
$prod$	$::=$	$\bigsqcap \overline{e}$	произведение
e	$::=$	v	переменная
	$\mid$	l	элемент решетки
sol	$::=$	$v_i \to l$	решение

Рис. 5: Синтаксис уравнений

l	$::=$	$\bot$	нижний элемент
	$\mid$	`true`	константа `true`
	$\mid$	`false`	константа `false`
	$\mid$	$\bullet$	константа `null`
	$\mid$	$\overline{\bullet}$	ненулевая ссылка
	$\mid$	$\top$	верхний элемент

Рис. 6: Решетка для для вывода аннотаций `@NotNull` и `@Contract`

Правые части уравнений представляются в виде суммы произведений (sum of products, sop, аналог дизъюнктивной нормальной формы из булевой алгебры).

Решатель находит максимальное решение (максимальную неподвижную точку).

3.2 Построение уравнений по графу конфигураций

Построение уравнений при выводе аннотаций `@NotNull` и `@Contract` происходит по похожим сценариям – при построении графа конфигураций все узлы размечаются дополнительной информацией, релевантной для анализа. (В большинстве узлов ничего релевантного для анализа не происходит, – они размечаются своего рода заглушками – нейтральными элементами.) Эта информация носит локальный характер – достаточно проанализировать содержимое узла (конфигурацию и эффект исполнения инструкции). Будем называть метки с такой информацией *локальными метками*. Затем, когда граф конфигураций сконструирован и построены локальные метки, обходом снизу вверх (начиная с листьев) узлы графа размечаются *аккумулирующими метками*. Аккумулирующая метка для текущего узла вычисляется через комбинацию локальной метки узла и аккумулирующих меток дочерних узлов. По аккумулирующей метке корневого узла строится уравнение. Далее для

обозначения локальных и аккумулирующих меток используются δ и σ соответственно.

Для такого сценария построения уравнений по графу конфигураций необходимо задать три функции – Δ, Σ, $\mathcal{E}$:

- $\Delta(c)$ – функция для вычисления локальной метки по конфигурации c.
- $\Sigma(\delta, \overline{\sigma_i})$ – функция для вычисления аккумулирующей метки по локальной метке δ и списку аккумулирующих меток $\overline{\sigma_i}$ в дочерних узлах.
- $\mathcal{E}(\sigma)$ – конструирует по аккумулирующей метке σ корневого узла уравнение.

Для листовых узлов $\sigma = \delta$. То есть, Σ применяется только для нелистовых узлов.

3.3 Вывод уравнений для `@NotNull` параметров

Для того, чтобы определить анализ с помощью *Faba*, необходимо выполнить следующие шаги (некоторые низкоуровневые детали опущены).

1. Ввести необходимые для анализа абстрактные значения (подклассы `BasicValue`, см. детали в [10]), определить для новых значений отношения $\equiv$ (эквивалентность) и $\subseteq$ (вложение, `instanceOf`), определить интерпретатор над абстрактными значениями (прогонка в терминах суперкомпиляции). После этого *Faba* будет автоматически строить графы конфигураций для анализа.
2. Определить Δ, Σ, $\mathcal{E}$ для построения уравнений по графу конфигураций.

В случае вывода `@NotNull` аннотаций *Faba* строит точно такие же графы конфигураций, как и *Kanva-micro* (с точностью до того, что *Faba* строит повторяющиеся поддеревья графа конфигураций ровно один раз). Эта часть кратко рассмотрена в 2 и подробно разобрана в [10].

В данном разделе разбирается, как устроен вывод уравнений по построенному графу конфигураций для вывода `@NotNull` аннотаций для параметров. Вся сответствующая алгебра представлена на Рис. 7.

Синтаксис меток $label^+$ практически совпадает с синтаксисом rhs с Рис. 5: из всех элементов решетки используются только $\overline{\bullet}$ и $\top$, дополненные специальной заглушкой - `I`. Идентификатор m_i используется для

$$
\begin{array}{lcl}
label^+ & ::= & sop^+ \mid \overline{\bullet} \mid \top \mid \mathtt{I} \\
sop^+ & ::= & \bigsqcup \overline{prod^+} \\
prod^+ & ::= & \bigsqcap \overline{m_i}
\end{array}
$$

$\sqcup$	$\top$	s_2^+	$\mathtt{I}$	$\overline{\bullet}$
$\top$	$\top$	$\top$	$\top$	$\top$
s_1^+	$\top$	$s_1^+ \sqcup s_2^+$	s_1^+	s_1^+
$\mathtt{I}$	$\top$	s_2^+	$\mathtt{I}$	$\overline{\bullet}$
$\overline{\bullet}$	$\top$	s_2^+	$\overline{\bullet}$	$\overline{\bullet}$

$\sqcap$	$\top$	$\mathtt{I}$	s_2^+	$\overline{\bullet}$
$\top$	$\top$	$\top$	s_2^+	$\overline{\bullet}$
$\mathtt{I}$	$\top$	$\mathtt{I}$	s_2^+	$\overline{\bullet}$
s_1^+	s_1^+	s_1^+	$s_1^+ \sqcap s_2^+$	$\overline{\bullet}$
$\overline{\bullet}$	$\overline{\bullet}$	$\overline{\bullet}$	$\overline{\bullet}$	$\overline{\bullet}$

$$\Delta\left(\boxed{\texttt{return}}\right) = \top \qquad (\Delta_1)$$

$$\Delta\left(\boxed{\texttt{throw}}\right) = \overline{\bullet} \qquad (\Delta_2)$$

$$\Delta\left(\boxed{\bullet\texttt{!}}\right) = \overline{\bullet} \qquad (\Delta_3)$$

$$\Delta\left(\boxed{\circ\texttt{.m(}\bullet\texttt{)}}\right) = \boxplus m_{i\bullet} \qquad (\Delta_4)$$

$$\Delta\left(\boxed{\cdots}\right) = \mathtt{I} \qquad (\Delta_5)$$

$$\Sigma(\delta, \overline{\sigma_i}) = \delta \boxplus (\boxplus \overline{\sigma_i}) \qquad (\Sigma)$$

$$\mathcal{E}(\mathtt{I}) = (m' = \top) \qquad (\mathcal{E}_1)$$

$$\mathcal{E}(l) = (m' = l) \qquad (\mathcal{E}_2)$$

Рис. 7: Алгебра для вывода `@NotNull` параметров

обозначения i-ого параметра метода (и соответствующего аргумента в вызове метода) с сигнатурой m (сигнатура метода уникальна). В силу специфики анализа метка есть либо одна из констант $\overline{\bullet}$, $\top$, $\mathtt{I}$, либо сумма произведений (sop+) из переменных m_i.

Для меток $label^+$ определяются операции $\sqcup$ и $\sqcap$ – объединение и пересечение меток. Заглушка $\mathtt{I}$ является нейтральным элементом для обеих операций $\sqcup$ и $\sqcap$. Таким образом, $label^+$ представляет из себя решетку.

Правила для вычисления локальных меток представлены в схематично-мнемоническом виде. Для вычисления метки достаточно знать, какая инструкция байткода выполняется в данном узле и принадлежит ли данный узел к `null`-информированному поддереву (такие деревья обозначаются двойным контуром). В правилах в мнемоническом виде изображены инструкции байткода.

Правила Δ_1, Δ_2, Δ_3, Δ_4, Δ_5 расшифровываются следующим образом

- Δ_1. Для узла с инструкцией `RETURN`, `ARETURN`, `IRETURN`, ... локальная метка – верхний элемент решетки. Это соответствует нормальному завершению метода.

- Δ_2. Для узла с инструкцией `ATHROW`, находящегося внутри `null`-информированного дерева, локальная метка – $\overline{\bullet}$. Это соответствует инициализации исключения в случае, если параметр равен `null`.

- Δ_3. Если в узле происходит разыменовывание параметра (схематично обозначается как $\bullet$`!`, распознавание разыменовывания параметра по текущей конфигурации подробно описано в [10]), то локальная метка – $\overline{\bullet}$. Это соответствует тому, что при выполнений текущей инструкции возникает `NullPointerException` из-за нулевого значения параметра.

- Δ_4. Анализируемый параметр является аргументом вызова некоторого другого метода. Локальная метка – $\boxplus m_{i^\bullet}$, где m - сигнатура вызываемого метода, а $i^\bullet$ – индексы параметров, в которые передается $\bullet$. Например, если в узле происходит вызов `m(param, x, param)`, то локальная метка будет $m_1 \boxplus m_3$.

- Δ_5. Все остальные случаи в графе конфигураций не являются релевантными для анализа, локальная метка – заглушка `I`.

Правило Δ_4 является ключевым. Можно сказать, что в нем происходит совмещение анализа зависимостей и анализа нулевых ссылок, которые были разделены в *Kanva-micro*.

Комбинирование $\Sigma(\delta, \overline{\sigma_i})$ происходит просто – вначале объединяются дочерние аккумулирующие метки и результат пересекается с локальной меткой. Если аккумулирующей меткой корневого узла оказался нейтральный элемент `I`, правая часть уравнения – $\top$ (правило $\mathcal{E}_1$). В остальных случаях аккумулирующая метка корневого узла становится правой частью уравнения (правило $\mathcal{E}_2$).

Решением получившегося уравнения может быть либо $\overline{\bullet}$, либо $\top$. В итоге, параметр аннотируется как `@NotNull`, если $m_i \to \overline{\bullet}$.

```
class Examples {
  static void g(Object o, boolean b) {
    if (b) f(o, o);
    else   s(o, o);
  }

  static void f(Object o1, Object o2) {
    if (o1 == null) throw new NullPointerException;
    else s(o2, o2);
  }

  static void s(Object o1, Object o2) {
    t(o1); v(o2);
  }

  static void t(Object o) {
    o.toString();
  }

  static void v(Object v) {}
}
```

Рис. 8: Исходный код на Java.

a)
$$
\begin{aligned}
& f_1 = \overline{\bullet};\\
& f_2 = s_1 \boxplus s_2;\\
& g_1 = (f_1 \boxplus f_2) \boxplus (s_1 \boxplus s_2);\\
& s_1 = t_1;\\
& s_2 = v_1;\\
& t_1 = \overline{\bullet};\\
& v_1 = \top;
\end{aligned}
$$

b)
$$
\begin{aligned}
& f_1 \to \overline{\bullet};\\
& f_2 \to \overline{\bullet};\\
& g_1 \to \overline{\bullet};\\
& s_1 \to \overline{\bullet};\\
& s_2 \to \top;\\
& t_1 \to \overline{\bullet};\\
& v_1 \to \top;
\end{aligned}
$$

Рис. 9: a) Построенные уравнения; b) Решение.

3.4 Примеры

В листинге Рис. 8 приведен простой код на Java, чтобы на простейших графах конфигураций продемонстрировать, как вычисляются локальные и аккумулирующие метки для вывода `@NotNull` аннотаций для параметров. Соответствующие графы конфигураций для каждого анализируемого параметра ссылочного типа представлены в схематичном виде на Рис. 10. (Для экономии места графы изображены слева направо, а не сверху вниз. Вместо элементарных инструкций байткода, приведены соответствующие им фрагменты исходной программы. Низкоуровневые инструкции байткода – манипуляции с локальными переменными и стеком опущены для упрощения.) Полученная система уравнений и ее решение приведены на Рис. 9.

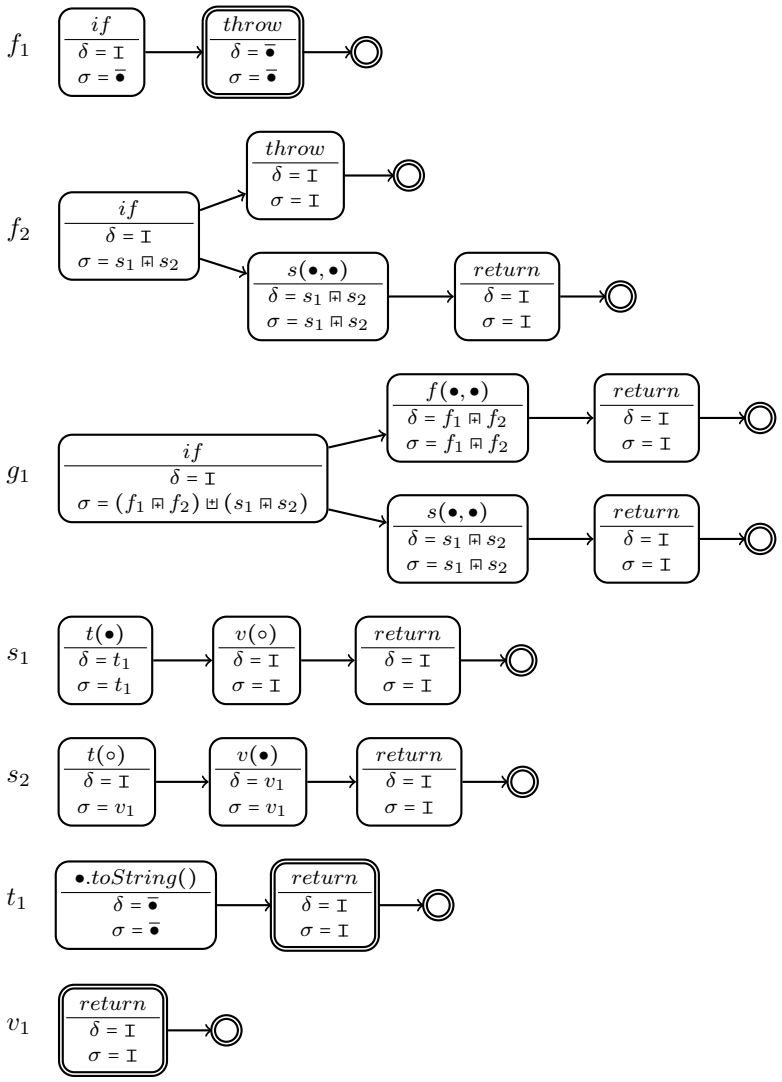

Рис. 10: Графы с локальными и аккумулирующими метками

Библиотека	#M[a]	#P[b]	IT[c]	IT_0[d]	ST[e]	#@[f]
JDK 1.7[g]	175066	131711	18.055	6.738	0.357	34577
commons-lang3-3.3.2	2911	2889	1.247	0.501	0.091	593
asm-5.0.1	1831	2343	0.833	0.404	0.027	462
commons-io-2.4	1188	1197	0.446	0.209	0.020	425
Android SDK 19	88109	62724	18.978	12.433	0.272	17709
Ceylon 1.0 compiler	11573	12788	2.537	1.243	0.077	4128
Eclipse 4.3 compler	6906	6369	3.259	1.541	0.051	1838
IDEA 13 (idea.jar)	170655	162756	13.868	5.063	0.332	63027

[a]Количество методов в библиотеке

[b]Количество параметров ссылочного типа

[c]Общее время индексирования (загрузка байткода, построение графа потока управления, анализ графа на приводимость, построение графа конфигураций, построение уравнений) в секундах

[d]"Чистое" время индексирования (построение графа конфигураций, построение уравнений) в секундах

[e]Время, потраченное на решение системы уравнений в секундах

[f]Количество выведенных `@NotNull` аннотаций для параметров

[g]Oracle JDK 1.7.0_45 (rt.jar)

Рис. 11: Результаты вывода `@NotNull` параметров

3.5 Результаты

Проект *Faba* [2] реализован на Scala. Результаты вывода `@NotNull` параметров для некоторых библиотек Java приведены на Рис. 11[1]. Как видно из таблицы, решение полученных уравнение требует на порядок меньше времени, чем индексация. В проекте *Faba* реализовано два решателя – простой и экономный. Экономный решатель переводит полученные уравнения в компактную форму (вся система уравнений упаковывается в однородный массив целых чисел). Дополнительной памяти не требуется – уравнения решаются in-place (массив обновляется деструктивно).

4 Заключение

Полученные результаты показывают, что описанный подход применим для автоматического аннотирования Java библиотек контрактами в средствах разработки (конкретно, в IntelliJ IDEA).

[1]Результаты получены на компьютере MacBook Pro с процессором i5 2.6GHz, OSX 10.9, Oracle JDK 1.8.0_05.

4.1 Близкие работы по анализу Java программ

Большинство существующих инструментов анализа байткода предназначены для поиска в нем ошибок или потенциально опасных мест или для верификации байткода. *Faba не анализирует* байткод на наличие ошибок. Основная цель описанного анализа – вывод свойств (аннотаций) в форме, пригодной для использования инструментами анализа кода внутри интегрированной среды разработки.

Основная цель технических решений *Faba* – сделать анализ пригодным для интеграции в средства разработки (модульность, быстродействие, минимальные требования к ресурсам по памяти).

Насколько можно судить по опубликованной литературе, у предложенного подхода нет прямых аналогов.

Наиболее близким по духу является анализ в анализаторе Julia [16] – в этой работе анализ нулевых ссылок делается с помощью булевой логики. Однако, принципиальным отличием является цель анализа. Julia анализирует байткод с целью найти в нем ошибки. Сравнить технические характеристики *Faba* и Julia не представляется возможным, так как Julia предлагается в виде сервиса – пользователь загружает свое Java приложение на сайт Julia, оно анализируется на на сервере и пользователю предоставляются результаты анализа.

Наиболее близким по техническим целям является работа [5], в которой авторы рассматривают подходы к эффективным и масштабируемым реализациям формальных методов на практике применительно к задаче аннотирования Java программ достаточными условиями завершаемости циклов (`for`-loops).

Решение для устранения избыточности графа конфигураций (которое будет описано в следующей статье цикла) основано на дополнительном анализе графа потока управления. Похожие оптимизации делаются в работе [4] для минимизации использования памяти верификатором байткода Java.

4.2 Анализ программ методами суперкомпиляции

В основе *Kanva-micro* и *Faba* – использование методов суперкомпиляции [17, 11] для анализа программ.

В большинстве ранних работ по применению суперкомпиляции для анализа программ практиковался простой трансформационный подход: использовался полноценный суперкомпилятор (делающий эквивалентные преобразования), однако вместо анализа исходной программы анализировалась суперкомпилированная программа, в которой некоторые избыточности исходной программы, затрудняющие анализ, были

устранены суперкомплятором. При таком подходе суперкомпилятор и анализ были разделены, и учет специфики анализа суперкомпилятором заключался, как правило, в выборе человеком подходящих настроек для суперкомпилятора (показательный пример – работа [12]).

Однако последние работы ([13, 9, 7, 6, 8, 14, 1]) показывают, что целесообразно создание специализированных проблемно-ориентированных суперкомпиляторов, учитывающих специфику проблемной области.

По сути, *Kanva-micro* и *Faba* являются проблемно-ориентированными суперкомпиляторами, учитывающими специфику анализа нулевых ссылок.

С точки зрения классических методов анализа потоков данных, ядро *Faba* (решатель уравнений над решетками и фреймворк для конструирования уравнений над решетками по графу конфигураций) является иструментом для конструирования уравнений для анализа потока данных и нахождения MOP (Meet Over all Paths) и MVP (Meet over Valid Paths) решений этих уравнений (см. [15], раздел 2.5.2). При решении задач анализа потока данных наиболее нетривиальной проблемой является перечисление соответствующих путей. В *Faba* для перечисления путей используются графы конфигураций.

4.3 Направление будущих работ

На данный момент в проекте *Faba* в анализаторах сделано некоторое количество упрощений, чтобы достичь модульности и производительности. С другой стороны, как показывают эксперименты, можно ввести дополнительные упрощения, которые дадут выигрыш в быстродействии и незначительную потерю точности анализа.

Направление будущих работ – поиск баланса между точностью и скоростью анализаторов с учетом специфики интегрированных средств разработки.

Список литературы

[1] A toolkit for building multi-result supercompilers. `https://github.com/ilya-klyuchnikov/mrsc`.

[2] Faba, FAst Bytecode Analysis of Java libraries. `https://github.com/ilya-klyuchnikov/faba`.

[3] Kanva-micro, the essence of supercompilation over abstract values. `https://github.com/ilya-klyuchnikov/kanva-micro`.

[4] C. Bernardeschi, N. De Francesco, and L. Martini. Efficient bytecode verification using immediate postdominators in control flow graphs. In *On The Move to Meaningful Internet Systems 2003: OTM 2003 Workshops*, pages 425–436. Springer, 2003.

[5] J. Fulara and K. Jakubczyk. Practically applicable formal methods. In *SOFSEM 2010: Theory and Practice of Computer Science*, pages 407–418. Springer, 2010.

[6] S. Grechanik. Inductive prover based on equality saturation for a lazy functional language. In *Proceedings of the Ershov Informatics Conference (PSI 2014)*. Springer, 2014.

[7] S. Grechanik, I. Klyuchnikov, and S. Romanenko. Staged multi-result supercompilation: Filtering by transformation. In *Forth International Workshop on Metacomputation in Russia*, 2014.

[8] A. Klimov. Solving coverability problem for monotonic counter systems by supercompilation. In *Proceedings of the Ershov Informatics Conference (PSI 2011)*. Springer, 2011.

[9] A. Klimov, I. Klyuchnikov, and S. Romanenko. Automatic verification of counter systems via domain-specific multi-result supercompilation. In *Third International Valentin Turchin Workshop on Metacomputation*, 2012.

[10] I. Klyuchnikov. Nullness analysis of java bytecode via supercompilation over abstract values. In *Forth International Workshop on Metacomputation in Russia*, 2014.

[11] I. Klyuchnikov and D. Krustev. Supercompilation: Ideas and methods. *The Monad Reader*, (23), 2014.

[12] A. Lisitsa and A. Nemytykh. Verification as a parameterized testing (experiments with the scp4 supercompiler). *Programming and Computer Software*, 33(1):14–23, 2007.

[13] G. E. Mendel-Gleason and G. W. Hamilton. Development of the productive forces. In *Third International Valentin Turchin Workshop on Metacomputation*, 2012.

[14] A. Nepeivoda. Turchin's relation and subsequence relation on traces generated by prefix grammars. In *Forth International Workshop on Metacomputation in Russia*, 2014.

[15] F. Nielson, H. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 2010.

[16] F. Spoto. Nullness analysis in boolean form. In *Software Engineering and Formal Methods (SEFM'08)*, pages 21–30. IEEE, 2008.

[17] V. F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(3):292–325, 1986.