УДК 004.4'23
С. А. Варивода, А. Д. Милакин, А. А. Солдатенков (2 курс, каф. ВТ, СПбГЭТУ “ЛЭТИ”)
С. В. Григорьев, (к. ф.-м. н., доцент кафедры информатики СПбГУ)

КОНТЕКСТНО ЧУВСТВИТЕЛЬНЫЙ АНАЛИЗ АЛИАСОВ В JAVA

Статическая проверка корректности, надёжности и других свойств программного обеспечения, а также его рефакторинг и реинжиниринг, практически невозможны без адекватных методов статического анализа кода. При этом, с развитием языков программирования, усложняется задача анализа создаваемого кода. Например, активное использование объектно-ориентированного программирования и соответствующих подходов к проектированию систем, делает всё более необходимым учёт контекста (переходов между методами и классами) при решении задач анализа кода, что является нетривиальной задачей. В 1995 году Томас Репс заметил, что многие задачи межпроцедурного статического анализа выразимы в терминах контекстно-свободной достижимости (КС-достижимость) в помеченных ориентированных графах [2], то есть в терминах существования пути между заданными вершинами, такого, что метки вдоль него образуют слово в некотором контекстно-свободном языке. Такая фундаментальная задача, как анализ алиасов, которая лежит в основе многих прикладных задач (taint analysis, nullability analysis) также выразима в терминах КС-достижимости [1,3]. Целью работы является разработка решения для контекстно чувствительного анализа алиасов для языка программирования Java на основе КС-достижимости.

Решение любой задачи в терминах КС-достижимости требует построения графового представления программы и некоторой КС грамматики. Построение графа программы выполняется в два этапа. На первом этапе строится граф каждого метода, который представляет из себя набор вершин ссылок на данные (*An*, где *A* — имя переменной, *n* — местоположение переменной, например номер строки в исходном коде) и вершин фактических данных (объект переданный из вне “*Si”* или новый объект “*On”*, где *n* — местоположение метода, а *i* — индивидуальный номер) соединенных ребрами с пометками данных, которые передают.

```
1.  class JurassicPark {
2.  private static final int MAX =  30;
3.
4.  Dinosaur[] data;
5.  int count;
6.
7.   JurassicPark() {
8.       Dinosaur[] t = new
Dinosaur[MAX];
9.       this.data = t;
10.  }
11.
12.  void add(Dinosaur d) {
13.     Dinosaur[] t = this.data;
14.     t[count++] = d;
15.  }
16.
17.  Dinosaur get(int idx) {
18.     Dinosaur[] t = this.data;
19.     Dinosaur p = t[idx];
20.     return p;
21.   }
22.  }
23.  class Dinosaur{
24.         String name; }
25.  class Main {
26.  static void main(String[] args) {
27.     JurassicPark p1 = new
JurassicPark();
28.     Dinosaur q = new Dinosaur();
29.     String t = "Ivan";
30.     q.name = t;
31.     p1.add(q);
32.     JurassicPark p2 = new
JurassicPark();
33.     Dinosaur g = p1.get(0);
34.     String m = g.name;
35.     System.out.println(m);
36.     String pav = t;   }
```

Рисунок 1. Код программы.

Рассмотрим пример на основе кода метода main приведённого на рисунке 1. Новый объект может создаваться через ключевое слово “new” (строка 27) или через присвоение литерала (строка 29). Во всех остальных случаях объект считается переданным из вне.
К таким объектам относятся переданные параметры методу, результат выполнения метода и передача данных объекта класса (строки 33, 34 соответственно). В последнем случае на ребре ставиться пометка с именем данных, которые передают.

Затем добавляются ребра, показывающие взаимодействие данных внутри метода. Они могут быть двух видов: без пометок (переход данных от переменных к переменной, строка 36), с пометками (переход данных объекта или сохранение данных в объект, строка 34 и 30). Последние проводятся только от вершинами фактических данных.

Одновременно с этим объединим все пометки, описанные выше, в нетерминал *road*: $<road> \rightarrow name \mid data \mid arr\_elem$, где *name* и *data* данные внутри класса, а *arr_elem* обозначение передачи элемента массива.

На втором этапе графы каждого метода соединяется в межпроцедурный граф рёбрами с пометками входа и выхода ($entry_N$ и $exit_N$ соответственно, где N — местоположение входа и выхода из метода). При этом проводятся рёбра от всех параметров всех вызовов метода к параметрам самого метода (результат для рассматриваемого кода представлен на рисунке 2) и в грамматику добавляется пара нетерминалов, выражающих скобочную последовательность:

$$<(_i> \rightarrow exit_N \mid entry_N^{-}$$
$$<)_i> \rightarrow entry_N \mid exit_N^{-}$$

Здесь $entry_N^{-}$ и $exit_N^{-}$ — обратные пути для $entry_N$ и $exit_N$ соответственно, а i — порядковый номер нетерминала. В нашем примере вызов метода *get* объекта *p1* на строке 33 породит такую пару нетерминалов:

$$<(_4> \rightarrow exit33 \mid entry33^{-}$$
$$<)_4> \rightarrow entry33 \mid exit33^{-}$$

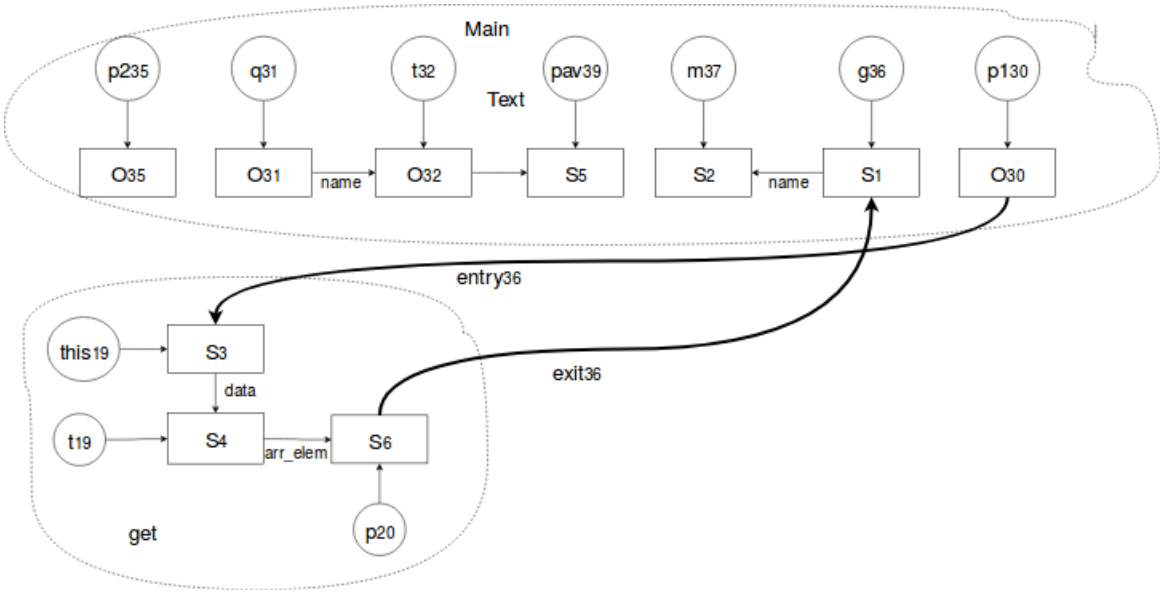

Рисунок 2. Граф программы.

Описав вход и выход из каждой функции, соединяем нетерминалы в общее правило и добавляем стартовый нетерминал <*memAlias*>:

$$<C> \rightarrow <(_i> <C> <)_i> \mid <C> <C> \mid <road> \mid <road^{-}> \mid Epsilon$$
$$<memAlias> \rightarrow <road^{-}> <memAlias> <road> \mid <memAlias> <memAlias> \mid <C>,$$

где *Epsilon* — пустой переход и $road^{-}$ — обратные пути для *road*.

Решение задачи КС-достижимости по построенным данным производится с помощью расширения библиотеки Meerkat [4], позволяющего использовать парсер комбинаторы в задачах КС-достижимости. Это позволяет упростить динамическое построение грамматики, так как становится возможным описать процесс построения грамматики в виде функции, параметризованной набором различного вида скобок, которую достаточно применить к набору для конкретного входного графа, чтобы получить готовый парсер. При использовании других подходов для этого потребовалось бы каждый раз генерировать новую грамматику, а затем по ней генерировать парсер, что выглядит инженерно более сложным решением.

В результате работы данного расширения получается структура данных, хранящая сжатое представление всех путей (Shared Packed Parse Forest, SPPF), удовлетворяющих заданным ограничениям. Для нашей задачи это будет, например, путь из O_{27} в S_2. По рис. 1 можно увидеть, что $p1_{27}$ и m_{34} алиасы. Важной особенностью используемого расширения является то, что кроме факта достижимости можно получить и сам путь, что может быть полезно для дальнейшего анализа.

В результате работы было создано консольное решение, позволяющее выполнять контекстно чувствительный анализ алиасов для языка программирования Java: реализован алгоритм построения графа и формирования контекстно свободной грамматики. Также реализован анализ SPPF для итогового анализа алиасов и нахождения пути появления алиаса. В дальнейшем планируется интеграция данного решения в среду разработки IntelliJ IDEA и создание на его основе решений для таких прикладных задач, как taint analysis или nullability analysis. Кроме того, необходимо провести экспериментальное исследование полученного решения на реальных проектах для того, чтобы оценить его практическую значимость и сравнить с другими существующими решениями для межпроцедурного анализа кода.

Литература.

1. Yan D., Xu G., Rountev A. Demand-driven context-sensitive alias analysis for Java //Proceedings of the 2011 International Symposium on Software Testing and Analysis. – ACM, 2011. – С. 155-165.
2. Reps T., Horwitz S., Sagiv M. Precise interprocedural dataflow analysis via graph reachability //Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages. – ACM, 1995. – С. 49-61.
3. Zhang Q. et al. Efficient subcubic alias analysis for C //ACM SIGPLAN Notices. – ACM, 2014. – Т. 49. – №. 10. – С. 829-845.
4. Сайт расширения Meerkat для решения задач КС-достижимости: https://github.com/YaccConstructor/Meerkat (Дата доступа: 28.03.2018)