Использование различных представлений java-программ для статического анализа

Е.А. Карпулевич <karpulevich@ispras.ru>
Институт системного программирования РАН,
109004, Россия, г. Москва,
ул. А. Солженицына, дом 25.

Аннотация. Статический анализ исходного кода используется для автоматизированного обнаружения дефектов программного обеспечения. Особо ощутима польза статического анализа при разработке больших проектов, состоящих из сотен тысяч строк кода, поскольку такой объем кода практически невозможно проверить вручную.
Статический анализатор, в отличие от компилятора, не так сильно ограничен по времени. Благодаря этому, можно реализовать более сложные и точные алгоритмы, которые выдают больше истинных и меньше ложных срабатываний, чем алгоритмы анализа компилятора. В основе работы любого
алгоритма лежит внутреннее представление программного кода. В статье рассматриваются различные варианты внутреннего представления программ и детекторы программных ошибок, работающие на этих представлениях. Анализ внутреннего представления в виде абстрактного синтаксического дерева (АСТ) позволяет быстро обнаруживать несложные ошибки, например опасное преобразование типов. С помощью абстрактного синтаксического дерева удобно искать ошибки, связанные с повторным использованием кода. Анализ графа потока управления (ГПУ) позволяет находить более сложные ошибки, для обнаружения, которых требуется проход по коду программы. Вместо прохода по коду анализ выполняется с помощью обхода ГПУ. С помощью анализа на ГПУ можно обнаружить такие дефекты, как, например, утечка ресурса, повторное освобождение ресурса, переполнение буфера.
Существуют и другие внутренние представления, на которых удобно проводить определенные классы анализов. В статье, в качестве примера, описаны принципы работы нескольких детекторов анализатора SVACE на соответствующих внутренних представлениях.

Ключевые слова: статический анализ, java, FindBugs, SVACE

DOI: 10.15514/ISPRAS-2015-27(6)-10

Для цитирования: Карпулевич Е.А. Использование различных представлений java-программ для статического анализа. Труды ИСП РАН, том 27, вып. 6, 2015 г., стр. 151-158. DOI: 10.15514/ISPRAS-2015-27(6)-10.

1. Введение

В настоящее время программный код в больших проектах исчисляется миллионами строк кода. Неудивительно, что в таких проектах присутствует огромное количество ошибок разной степени критичности. Обнаружить среди миллионов строк несколько строк кода, содержащих дефект, непросто, а отследить утечку ресурса или возможное разыменование нулевого указателя вручную иногда практически невозможно. Для автоматизированной проверки программ и выдачи предупреждений об ошибках необходим инструмент, специализирующийся на поиске ошибок - статический анализатор. Для языка программирования Java существует несколько известных статических анализаторов: FindBugs[1], SVACE[2], Jlint[3] и др.

Во время работы компилятор также может выдавать большое количество предупреждений о потенциальных дефектах, однако, большое количество из них не приводят к сбоям в работе программы (например, когда в новой версии компилятора функция объявлена устаревшей). Кроме того, анализ производимый во время компиляции ограничен по времени и ресурсам, так как основная задача java-компилятора - предоставить байткод максимально быстро.

Для получения адекватных предупреждений об ошибках при анализе программ важны быстрые и корректные алгоритмы поиска ошибок, продуманная архитектура анализатора и информативное промежуточное представление исходного кода программы, на котором строится весь анализ[4].

Различные промежуточные представления подходят для выявления в коде программы определенных типов ошибок. В статье проводится сравнение нескольких внутренних представлений в контексте статического анализа.

2. Получение различных внутренних представлений

Построение промежуточных представлений статического анализатора схоже с построением промежуточных представлений при компиляции исходного кода. На этапе лексического анализа исходного кода компилятор разбирает исходный код на последовательность лексем, из которых формирует абстрактное синтаксическое дерево. После этого происходят оптимизации на уровне абстрактного синтаксического дерева. На этапе кодогенерации по абстрактному синтаксическому дереву генерируется java-байткод. Для проведения анализа программы и дальнейшей её оптимизации по байткоду строится граф потока управления, граф вызовов.

Для статического анализа программ используются все перечисленные представления. На основе анализа байткода работают детекторы поиска

ошибок по шаблонам, на абстрактном синтаксическом дереве – детекторы клонов кода и поиск ошибок несоответствия отступов. Построение графа потока управления не является необходимым этапом статического анализа. Но существует класс ошибок, которые неудобно искать, анализируя абстрактное синтаксическое дерево. Такие ошибки как утечки памяти и разыменование нулевых указателей позволяет обнаружить анализ графа потока управления.

3. Поиск ошибок через анализ байткода и абстрактного синтаксического дерева.

Большая часть предупреждений FindBugs является результатом работы алгоритмов поиска ошибок на байткоде по характерным шаблонам. С помощью библиотеки ASM[5] происходит обход инструкций байткода.

Библиотека ASM позволяет работать с байткодом, она позволяет считывать байткод, а также предоставляет возможность по генерации и модификации байткода классов на лету.

Простой пример поиска по байткоду – детектор FindBugs ICAST_INTEGER_MULTIPLY_CAST_TO_LONG. Этот детектор проверяет на возможное переполнение типа integer до расширения до long. Такая ошибка содержится, например, в следующем фрагменте кода:

*long convertDaysToMilliseconds(int days) { return 1000*3600*24*days; }*

Чтобы найти ошибку такого рода достаточно проверить наличие в байткоде последовательности из двух инструкций IMUL (умножение двух целых) I2L(преобразование в long). Исправленный код может выглядеть так:

*long convertDaysToMilliseconds(int days) { return 1000L*3600*24*days; }*

Сигнатурный поиск по байткоду позволяет найти достаточно простые ошибки, такие как проверка знака битовой операции и ошибка форматирования строки и пр. Алгоритмы поиска шаблонов таких ошибок хорошо работают на внутреннем представлении в виде последовательности команд байткода. Но существуют ошибки, для поиска которых анализ байткода не является достаточным или является крайне неэффективным. Например, для поиска ошибок повторного использования (клонов) кода лучше подходит абстрактное синтаксическое дерево. Для работы алгоритмов поиска утечек ресурсов необходима информация, которую можно восстановить из байткода, но которая в нем не содержится: где ресурс выделяется и освобождается, функция какого объекта вызывается. Также, для поиска утечек ресурсов требуются пользовательские спецификации, полная или частичная девиртуализация и межпроцедурный анализ.

Алгоритмы, реализованные на байткоде можно успешно реализовать на абстрактном синтаксическом дереве, однако реализация алгоритмов может стать значительно сложнее.

Рассмотрим детектор ICAST_INTEGER_MULTIPLY_CAST_TO_LONG. Для того чтобы найти ошибку переполнения в абстрактном синтаксическом дереве

необходимо проверять типы, значения и эмулировать арифметические операции, требуется значительно больше усилий.

Задачи из класса обнаружения клонов кода наоборот проще решаются на основе анализа абстрактного синтаксического дерева или анализа графа программных зависимостей[6]. Например, поиск ошибки повторного использования на байткоде затруднен необходимостью восстанавливать структуру условных операторов и операторов цикла, отсутствием информации об отступах и позициях идентификаторов.

Ошибки повторного использования возникают при дублировании кода копированием с внесением последующих изменений. Один из вариантов такой ошибки – копирование условного оператора с последующей заменой одной переменной на другую. Иногда программист успешно меняет копию фрагмента кода в пяти местах, а в одном забывает.

Чтобы найти ошибку повторного использования необходимо проверить похожесть двух фрагментов кода и провести анализ идентификаторов на наличие неполных замен. Замена идентификатора A будет неполной если везде, кроме одного места, идентификатор A заменен на идентификатор B.

Кроме того, необходимо каким-то образом выявить похожие фрагменты кода, на которых и будет работать алгоритм. Для поиска похожих фрагментов кода в лоб можно использовать суффиксные деревья для поиска двух одинаковых подстрок в последовательности лексем, полученных из исходного кода. Но такой поиск достаточно медленный (сложность построения суффиксного дерева O(n), проверка похожих частей O(m)), так как поиск идет по всему коду файла.

Около 40% случаев копирования кода – копирование базовых блоков и функций[7]. Программист копирует небольшую функцию, условный оператор целиком, одно из условий условного оператора, заголовок цикла или весь цикл, заменяя один или несколько идентификаторов. К тому же чаще всего вставка происходит рядом копированием. Полная информация о таких фрагментах кода и о порядке их следования содержится в абстрактном синтаксическом дереве. В статическом анализаторе SVACE поиск повторного использования кода реализован через поиск похожих частей кода на абстрактном синтаксическом дереве (сложность обнаружения фрагментов для проверки O(1), проверка похожих частей O(m)) с последующим анализом на повторное использование. В поиске повторного использования в анализаторе SVACE учитывается не только похожесть соседних фрагментов кода, но и соответствие отступов строк в них.

4. Анализ графа потока управления

Для поиска ошибок разыменования нулевого указателя или поиска утечки ресурса требуется анализировать пути исполнения программы. В таком случае можно использовать граф потока управления. Алгоритмы поиска ошибок на графе потока управления похожи на алгоритмы компиляторного анализа.

Анализ графа потока управления используется в статическом анализаторе SVACE, что позволяет анализатору находить утечки ресурсов. Для корректной работы алгоритма поиска утечек необходима возможность хранения и передачи межпроцедурной информации, так как часто ресурс выделяется в одной функции, а используется и освобождается в других. В статическом анализаторе SVACE после анализа функции составляется и сохраняется её аннотация (информация о поведении функции). Кроме того пользователю необходимо указать какие функции выделяют, а какие закрывают ресурсы. Для этих целей в SVACE существует возможность добавлять пользовательские спецификации, описывающие поведение функций.

Поиск утечки ресурса идет в процессе обхода графа управления. Фиксируются моменты выделения и освобождения ресурса и, если было выделение ресурса а после не было его освобождения, выдается предупреждение об утечке.

В языке java все нестатические неприватные (то есть, protected, package и public) методы являются виртуальными. Для анализа графа потока управления на предмет ошибок нужно понимать метод какого класса вызывается в коде. Провести точную девиртуализацию не всегда представляется возможным. В таком случае можно прибегнуть к частичной девиртуализации. При частичной девиртуализации вместо одного кандидата методу соответствует некоторый набор кандидатов. Точность анализа при частичной девиртуализации для ряда детекторов снижается, но и время необходимое для девиртуализации сокращается. Понимать какой метод вызывается нужно, в том числе, для поиска утечек. В java возможна ситуация, когда базовый класс не выделяет ресурс в отличие от своих потомков. В этом случае необходима девиртуализация, чтобы понять был ли выделен ресурс.

5. Поиск ошибок в многопоточных программах

Многие программы выполняются в несколько потоков с использованием примитивов синхронизации. При таком выполнении в процессе работы программы могут возникать взаимные блокировки или состояние гонки. Для задачи поиска взаимных блокировок граф потока управления является слишком громоздким и содержит информацию, которая не пригодится для поиска ошибок синхронизации. В инструменте Jlint много внимания уделили многопоточности. В качестве внутреннего представления Jlint использует граф зависимостей блокировок[8], для того чтобы алгоритмы поиска ошибок синхронизации были проще и точнее.

6. Объединение представлений

Существуют и другие внутренние представления программ, например граф программных зависимостей и граф вызовов. Они подходят для других задач, например, граф программных зависимостей удобен для поиска клонов кода.

Наличие нескольких представлений, на которых можно реализовать анализы различных классов ошибок заставляет задуматься о создании статического анализатора с несколькими внутренними представлениями. И действительно, многие анализаторы работают с несколькими представлениями. Например, статический анализатор SVACE ищет ошибки с помощью алгоритмов работающих на представлениях в виде байткода, графа вызовов, графа потока управления и др.

Кроме простого наличия нескольких представлений важна возможность безболезненного перехода от одного внутреннего представления к другому. При наличии такого перехода возможно не только повысить информативность навигации и сообщений об ошибках, но и делать несколько анализов на предмет одной ошибки на различных представлениях, а потом объединять результаты.

Очень важна связь между исходным кодом и внутренними представлениями программы. В процессе компиляции исходного кода в байткод происходит потеря некоторой части полезной информации о программе, например, полностью пропадает информация о наличии и количестве пробельных символов. Анализ ошибок такого рода можно проводить только на этапе компиляции.

В компиляторе javac есть внутреннее представление исходного кода в виде абстрактного синтаксического дерева. В таком случае достаточно интересной выглядит идея построить статический анализатор кода на основе компилятора, добавив в него несколько дополнительных внутренних представлений и реализацию алгоритмов для поиска ошибок.

В код компилятора javac успешно добавлены детекторы, работающие на абстрактном синтаксическом дереве, обнаруживающие ошибки повторного использования, ошибки выбора объекта для синхронизации, одинаковые ветки в условном или тернарном операторе. Полнота (отношение количества обнаруженных ошибок к общему количеству предупреждений) анализа для этих детекторов составила около 80%.

Список литературы

[1]. FinBugs – http://findbugs.sourceforge.net/findbugs2.html
[2]. В.П. Иванников, А.А. Белеванцев, А.Е. Бородин, В.Н. Игнатьев, Д.М. Журихин, А.И. Аветисян, М.И. Леонов. Статический анализатор Svace для поиска дефектов в исходном коде программ. Труды Института системного программирования РАН Том 26. Выпуск 1. 2014 г. Стр. 231-250.
[3]. Cyrille Artho. Finding faults in multi-threaded programs. March 15, 2001. (http://artho.com/jlint/mthesis.pdf)
[4]. Nick Rutar, Christian B. Almazan, Jeffrey S. Foster. A Comparison of Bug Finding Tools for Java. (http://www.cs.umd.edu/~jfoster/papers/issre04.pdf)
[5]. ASM framework – http://asm.ow2.org/index.html

[6]. Sevak Sargsyan, Shamil Kurmangaleev, Vahagn Vardanyan, Vachagan Zakaryan. Code Clones Detection Based on Semantic Analysis for JavaScript Language. October 1, 2015 (https://csit.am/2015/9a.html)
[7]. Zhenmin Li, Shan Lu, Suvda Myagmar and Yuanyuan Zhou. CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code. (http://opera.ucsd.edu/paper/OSDI04-CPMiner.pdf)
[8]. Jurgen Graf, Martin Hecker, Martin Mohr, and Benedikt Nordhoff. Lock-sensitive Interference Analysis for Java: Combining Program Dependence Graphs with Dynamic Pushdown Networks. 2013. (https://pp.ipd.kit.edu/uploads/publikationen/pdgwithdpn2013id.pdf)

Using Different Views Java-Programs for Static Analysis

E.A. Karpulevitch <karpulevich@ispras.ru>
Institute for System Programming of the RAS,
25 Alexander Solzhenitsyn Str., Moscow, 109004, Russian Federation

Absract. Static analysis of the source code used for the automated detection of software defects. Particularly noticeable benefits of static analysis in the development of large projects, consisting of hundreds of thousands of lines of code, because this amount of code is almost impossible to check manually.
Static analyzer of the compiler in contrast, not so much limited in time. Because of this, you can implement more complex and accurate algorithms that give more truth, and less false positives than the compiler's analysis algorithms. At the heart of any algorithm is an internal representation of the program code. The article discusses the various options for the internal representation of programs and software bug detectors that work on these ideas. Analysis of the internal representation of an abstract syntax tree (AST) allows you to quickly detect simple errors, such as a dangerous type conversions. By using abstract syntax tree is convenient to look for errors associated with re-use of code. An analysis of the control flow graph (CFG) allows you to find a more sophisticated error detection which requires passage by the program code. Instead pass code analysis is executed using the CFG bypass. Through analysis of the CFG can detect defects such as, for example, a resource leak, double release of the resource, buffer overflow. There are also other internal representations, which is convenient to carry out certain tests classes. The article, by way of example, the principles of operation described SVACE analyzer several detectors corresponding internal representations.

Keywords: static analysis, java, FindBugs, SVACE

DOI: 10.15514/ISPRAS-2015-27(6)-10

For citation: Karpulevitch E.A. Using Different Views Java-Programs for Static Analysis. *Trudy ISP RAN/Proc. ISP RAS, vol. 27, issue 6, 2015,* pp. 151-168 (in Russian). DOI: 10.15514/ISPRAS-2015-27(6)-10

References

[1]. FinBugs – http://findbugs.sourceforge.net/findbugs2.html
[2]. V.P. Ivannikov, A.A. Belevancev, A.E. Borodin, V.N. Ignat'ev, D.M. Zhurikhin, A.I. Avetisjan, M.I. Leonov. Staticheskij analizator Svace dlja poiska defektov v iskhodnom kode programm [Svace: static analyzer for detecting of defects in program source code] Trudy ISP RAN [The Proceedings of ISP RAS], volume 26, issue 1, pp. 231–250. DOI: 10.15514/ISPRAS-2014-26(1)-7. (in Russian)
[3]. Cyrille Artho. Finding faults in multi-threaded programs. March 15, 2001. (http://artho.com/jlint/mthesis.pdf)
[4]. Nick Rutar, Christian B. Almazan, Jeffrey S. Foster. A Comparison of Bug Finding Tools for Java. (http://www.cs.umd.edu/~jfoster/papers/issre04.pdf)
[5]. ASM framework – http://asm.ow2.org/index.html
[6]. Sevak Sargsyan, Shamil Kurmangaleev, Vahagn Vardanyan, Vachagan Zakaryan. Code Clones Detection Based on Semantic Analysis for JavaScript Language. October 1, 2015 (https://csit.am/2015/9a.html)
[7]. Zhenmin Li, Shan Lu, Suvda Myagmar and Yuanyuan Zhou. CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code. (http://opera.ucsd.edu/paper/OSDI04-CPMiner.pdf)
[8]. Jurgen Graf, Martin Hecker, Martin Mohr, and Benedikt Nordhoff. Lock-sensitive Interference Analysis for Java: Combining Program Dependence Graphs with Dynamic Pushdown Networks. 2013. (https://pp.ipd.kit.edu/uploads/publikationen/pdgwithdpn2013id.pdf)