УДК 004.41

М.В. Зубов, А.Н. Пустыгин, Е.В. Старцев

Математическое моделирование универсальных многоуровневых промежуточных представлений для статического анализа исходного кода

Для выполнения статического анализа было предложено использовать универсальные многоуровневые промежуточные представления. Для оценки их эффективности и определения критериев качества была разработана общая математическая модель промежуточного представления. На ее основе были получены математические модели для универсального представления уровня классов и потока управления. Было выполнено численное моделирование статического анализатора, позволяющее оценить эффективность использования таких промежуточных представлений на практике.

Ключевые слова: программирование, статический анализ, промежуточное представление, математическая модель, исходный код, рефакторинг, Java.

Статический анализ служит для автоматизации задач разработки программного обеспечения. Он позволяет выполнять поиск ошибок, улучшать качество кода, делать подсказки по рефакторингу, извлекать информацию из исходного кода. Достоинства применения универсальных промежуточных представлений при статическом анализе исходного кода были освещены в [1]. В качестве универсальных промежуточных представлений выступили 2 разработанных представления – универсальное классовое представление UCR и универсальное представление потока управления UCFR. Необходимо количественно оценить эффективность использования данных представлений. Для этого необходимо формализовать критерии эффективности и провести численное моделирование.

Получение промежуточных представлений. Основным способом перевода исходного кода из текстового формата является синтаксический анализ [2]. Исходный код является текстом на формальном языке – языке программирования. Такой текст легко представляется в виде деревьев (графов без циклов) разбора согласно его формальной грамматике в виде абстрактного синтаксического дерева разбора [3]. Так как эффективной, изученной и неоднократно используемой формой представления исходного кода является дерево, то будем рассматривать собственные разрабатываемые представления как деревья.

Таковыми является большинство используемых представлений в различных анализаторах [4, 5]. Большинство анализаторов просто использует AST, например Pylint [6] или PMD [7]. Другой вариант представлений, например представление в виде реляционной базы данных, предложенное в проекте Omega [8], хранит данные в виде *B*-деревьев и фактически является табличным представлением AST. Если рассмотреть текст на логическом языке программирования в качестве представления, например из анализатора ASTLOG [9], то база знаний на таком языке является деревом, а сам текст на таких языках также можно представить в виде дерева разбора по формальной грамматике. Представление метамоделей в виде псевдокода из Moose [10] также можно считать деревом, так как такой псевдокод соответствует некоторой формальной грамматике и может быть разобран в виде дерева.

Модель промежуточного представления как дерева. Построим универсальную модель дерева. В полном дереве можно выделить конечное количество поддеревьев. Например, если рассматривать корневой узел, то он будет иметь множество своих поддеревьев, вложенных в него. Каждое из этих поддеревьев может иметь свои вложенные и т.д. Пусть $A_i \in A$ – некоторое поддерево из множества таких поддеревьев – A. Тогда A_0 – все дерево целиком, так как является поддеревом, содержащим корень. $A_1, A_2, ..., A_m$ – поддеревья, $m > 0$. Каждое поддерево можно разделить на корень, множество поддеревьев и множество оконечных «листьев» – элементов, от которых нет нисходящих связей. Таким образом, любое поддерево, включая корневое, можно описать следующим образом:

$$A_i = (o, AS_i, \mathrm{L_i}), \quad (1)$$

где $o \in O$ – узел дерева из множества возможных, $AS_i \subset A$ – множество поддеревьев для текущего корня, $L_i \subset L$ – множество листьев текущего корня, L – множество всех возможных листьев.

По аналогии с формулой (1) можно составить модель, описывающую любое промежуточное представление, которое является деревом. Ранее было предложено использовать универсальные многоуровневые промежуточные представления [5]. Были разработаны универсальное классовое представление [5] и универсальное представление потока управления [11].

Так как используемые представления – деревья, то задачи анализа сводятся к задачам обхода дерева. Сложность такой задачи зависит от глубины вложенности и степени ветвления дерева. При использовании высокоуровневых представлений сокращается степень ветвления дерева. Для увеличения быстродействия анализа представления в нем также должна быть сокращена глубина вложенности. В разработанных представлениях ее можно ограничить конкретной величиной, разбив представление на уровни. Построим математические модели для представлений уровня классов и потока управления.

Модель классового представления. Классовое представление имеет более жесткую структуру с четко разделенными уровнями. Потому для обозначения поддеревьев модели имеет смысл ввести двойную индексацию. Каждое поддерево будем обозначать как $A_{i,j}$, где i – номер поддерева на уровне, а j – номер уровня. Будем использовать основную модель представления (1). Необходимо сразу ограничить набор вершин, которые будут использоваться в представлении. Это классы, их родители, поля, методы.

Возможные вершины представления: p – project, c – class, r – parent, f – field, m – method, g – arg, $O = \{p,c,r,f,m,g\}$. В таком представлении допустимо всего 4 уровня: уровень проекта, уровень класса, уровень структуры класса и уровень структуры метода. Таким образом, можно расписать поддеревья представления по уровням. Листья, обозначающие имена классов и идентификаторы, будут входить во множество L. Запишем каждый уровень в виде модели:

$$A_{i,0} = (r, AS_{i,1}, \{\}), \quad A_{i,1} = (c, AS_{i,2}, LC_i),$$

где $lc_{i,1}$ обозначает id, а $lc_{i,2}$ – name, $lc_{i,1}, lc_{i,2} \in LC_i$;

$$A_{i,2} = (o_i, AS_{i,3}, LV_i),$$

где $o_i \in \{r,f,m\}$, а $lv_{i,j} \in LV$ характеризует конкретный корень поддерева (родитель, поле или метод);

$$A_{i,3} = (g, \{\}, LG_i),$$

где $\mathrm{lg}_{i,1}$ обозначает name, а $\mathrm{lg}_{i,2}$ – type, $\mathrm{lg}_{i,1}, \mathrm{lg}_{i,2} \in LG_i$.

Модель представления графа потока управления. Представление графа потока управления будет описывать только конкретный участок выполнения программы – функцию или метод класса. Сам по себе граф потока управления является ориентированным графом общего вида, допускающим циклы. Чтобы привести его в более удобный вид, будем хранить дуги графа потока управления отдельными элементами, указывая, от какого блока к какому идет дуга (подход основан на GraphML [12]). Для удобства отображения и обработки будем считать условные операторы в качестве отдельных вершин. Это отличается от классической модели потока управления, где операции перехода отображаются только в виде ребер (дуг) графа [13]. Отображение такого графа будет больше соответствовать блок-схеме алгоритма, а также позволит хранить и анализировать условия, по которым выбирается направление выполнения программы.

Учитывая все возможные ветвления в графе потока управления, можно ограничить его множество вершин. $O = \{p,m,f,b,l,t,y,r,i,w,h\}$, где p – project, m – method, f – function, b – block, l – flow, t – try/except, y – finally, r – for, i – if, w – while, h – with. В таком представлении допустимо всего 3 уровня: уровень проекта, уровень функций и методов, уровень описания потока управления (вершины и переходы между ними). Основные допустимые вариации могут быть только на третьем уровне, на котором описывается сам поток управления. Он может состоять из различных типов блоков. Первый уровень не имеет листьев, а третий не имеет поддеревьев, что логично, так как модель ограничена в глубину. Таким образом, можно составить его модель по уровням поддеревьев:

$$A_{i,0} = (p, AS_{i,1}, \{\}), \quad A_{i,1} = (o_{i,1}, AS_{i,2}, LF_i),$$

где $o_{i,1} \in \{m,f\}$, lf_i – имя функции или метода, $lf_i \in LF_i, LF_i \subset L$.

$$A_{i,2} = (o_{i,2}, AS_{i,3}, LB_i),$$

где $o_{i,2} \in \{b,l,t,y,r,i,w,h\}$, LB_{i} – характеристики конкретного узла (*id* – для блоков, from и to – для дуг графа).

Реализация анализа «Контроль разделения ответственности». Для выполнения задач извлечения информации из исходного кода и предложений по рефакторингу был составлен список анализаторов, использующий разработанные представления. Часть анализов из этого списка была реализована на языке Python. Кроме того, были реализованы генераторы преставлений для языков Java и Python на языках Java и Python соответственно. Исходный код опубликован в GitHUB под открытой лицензией [14]. Рассмотрим анализ «Контроль разделения ответственности». Он предлагает выполнить рефакторинг классов, перегруженных функциональностью, согласно описанию в [15]. Этот анализатор просматривает все классы проекта, ищет классы, перегруженные полями и методами, а также слишком сложные методы с большим количеством ветвлений в потоке управления. Результат работы – текстовый вывод в формате, аналогичном листингу 1. В нем показан пример анализа собственного кода и предложений по улучшению класса ru.csu.stan.java.classgen.automaton.ClassContext.

Листинг 1

Пример текстового вывода анализатора «Контроль разделения ответственности»

```
  Class ClassContext has potential problem with too many fields (20). Maybe
you should divide this class into some smaller?
  Class ClassContext has method processTypeTag() with too many flows (30).
Maybe you need to extract a new method?
  Class ClassContext has too many methods. Looks like it has too many re-
sponsibilities. Maybe you should divide it?
```

Для практической оценки эффективности использования универсальных высокоуровневых представлений необходимо было провести численное моделирование скорости анализа универсальных представлений и AST. Для этого, в свою очередь, необходимо было реализовать анализатор «Контроль разделения ответственности» на AST. Этот анализ будет основан на некоторых эвристиках, однако алгоритмически будет эквивалентен, так как заключается в полном обходе всего дерева представления для получения информации о классах и дополнительного обхода содержимого методов.

Согласно математической модели использование высокоуровневых представлений должно ускорить выполнение анализа и в целом изменить характер роста затрат на анализ при росте объема проекта. Таким образом, быстродействие анализаторов может служить критерием эффективности использования представлений. Можно теоретически оценить быстродействие на примере небольшого класса – ru.csu.stan.java.classgen.util. ClassIdGenerator из собственного кода и класса, который перегружен ответственностью, – ru.csu.stan.java.classgen.automaton. ClassContext. AST для первого класса содержит 195 узлов, из них 134 приходится на методы. Классовое представление содержит 33 узла, а поток управления – 47. То есть для анализа AST понадобится обойти 195 + 134 = 329 узлов, а для высокоуровневых представлений – 33 + 47 = 80, что в 4 раза меньше. AST для второго класса содержит 3352 узла, из них – 3015 приходятся на методы, UCR – 171, а UCFR – 364. То есть анализатору на AST придется обойти 3352 + 3015 = 6367 узлов, а анализатору высокоуровневых представлений 171 + 364 = 535, что почти в 12 раз меньше. То есть при анализе различных классов должна увеличиваться производительность от 4 до 12 раз.

Практическая оценка эффективности модели в задачах статического анализа. Для практической оценки эффективности по быстродействию анализатора, написанного на высокоуровневых представлениях, необходимо собирать статистические данные на большом количестве запусков утилит, чтобы исключить случайную составляющую. Кроме того, для исследования роста времени анализа необходимо анализировать несколько проектов. Для моделирования использовались собственные разработки и открытый компилятор языка Java из пакета OpenJDK [16]. Названия проектов и их основные характеристики показаны в табл. 1.

Так как работа утилит анализа включает в себя чтение входных XML-файлов с промежуточными представлениями, то сравнивать данные простых запусков не имеет смысла. Утилита, использующая AST, выполняет 1 дисковую операцию, а использующая UCR+UCFR – 2. Любая такая операция занимает в несколько раз больше времени и вносит большую случайную составляющую в исследуемые экспериментальные данные. Во избежание этого для эксперимента были созданы спе-

циальные цели запуска (отдельные исследовательские версии утилит), которые выполняли анализ заданное число раз, при этом данные с диска читались только один раз, перед первым выполнением, и в дальнейшем брались из оперативной памяти.

Таблица 1

Характеристики используемых проектов

Проект	Количество классов	Количество строк	Размер, кБ
Test	13	172	48
Jclassgen	32	3141	212
Jcfgen	51	4575	276
Javac	525	70343	3277

Также необходимо исключить все случайные составляющие, возникающие при работе утилиты на реальной машине. При этом необходимо выполнить запуск несколько раз, а в результате взять усредненные значения. Для выполнения экспериментов использовался компьютер с 4-ядерным процессором Inter Core i5 760 2,8 ГГц 8 МБ кэша, 8 ГБ оперативной памяти DDR 1333 МГц, жестким диском WD 500ГБ 5400rpm SATA1 с файловой системой ext4, операционная система – Gentoo Linux, ядро версии 3.10.25. Для выполнения использовались следующие версии ПО: JDK – 8u45, Python – 2.7.5, система собрана с GCC – 4.7.3.

В процессе работы измерялось общее время работы на нескольких запусках без учета времени, затрачиваемого на чтение с диска. Также фиксировалось количество этих запусков – n. В результате измерений получались следующие величины: n – количество запусков утилиты; T_{AST} – общее время, за которое выполнялась утилита, использующая AST заданное число раз; T_U – общее время, за которое выполнялась утилита, использующая универсальные высокоуровневые представления заданное число раз; $\Delta = (T_{\text{AST}} - T_Y)$ – разница во времени между запусками утилит; $\Delta_з = \frac{(T_{\text{AST}} - T_U)}{n}$ – разница во времени между одним запуском утилиты.

Тогда критериями эффективности использования представлений могут выступать величины времени исполнения одного анализа и процентный прирост производительности одного анализатора, относительно другого. То есть $t_{\text{AST}} = \frac{T_{\text{AST}}}{n}$ – среднее время одного запуска утилиты, использующей AST; $t_U = \frac{T_U}{n}$ – среднее время одного запуска утилиты, использующей универсальные высокоуровневые представления; $P_{\text{AST}} = \frac{\Delta}{T_{\text{AST}}} \times 100\%$ – процентный прирост производительности относительно AST.

Исследование величин t представляет наибольший интерес, так как именно они прогнозировались в исходных графиках. В качестве n было выбрано n = 10000, так как это заставляет выполняться утилиту достаточно долго, что позволяет уменьшить влияние случайных нагрузок, например фоновых задач операционной системы (уменьшить случайную составляющую). Результаты исследования показывают, что на AST анализ выполняется значительно дольше, до 7 ч (табл. 2).

Таблица 2

Данные о времени работы анализаторов

Проект	Общее время, мс		Δ, мс	$\Delta_з$, мс	P_{AST}, %
	U-R	AST			
Test	5519	49316	43797	4,38	88,81
Jclassgen	59562	762702	703140	70,31	92,19
Jcfgen	94745	1263926	1169181	116,92	92,50
Javac	2346677	25813860	23467183	2346,72	90,91

По табл. 2 можно рассчитать средние величины прироста производительности. $P_{\text{AST ср}} = 91{,}10\%$.

Как видно, при таком приросте производительности утилита выполняется быстрее в 10 раз, что является достаточно существенной величиной (1 порядок). Такая величина вполне согласуется с полученной теоретически, так как она высчитывается для всего проекта в среднем. Для расчета критериев среднего времени воспользуемся данными табл. 2 и величиной $n = 10000$. Результаты показаны в табл. 3.

Таблица 3

Среднее время выполнения одного анализа на разных промежуточных представлениях

Проект	U-R	AST
Test	0,55	4,93
Jclassgen	5,96	76,27
Jcfgen	9,47	126,39
Javac	234,67	2581,39

Как видно (табл. 3), величины среднего времени выполнения отличаются на порядок. По этим данным можно построить графическую зависимость, более наглядно демонстрирующую то, как расходятся величины (рис. 1). Эти данные на практике доказывают теоретическое предположение о высокой эффективности использования универсальных многоуровневых промежуточных представлений для статического анализа.

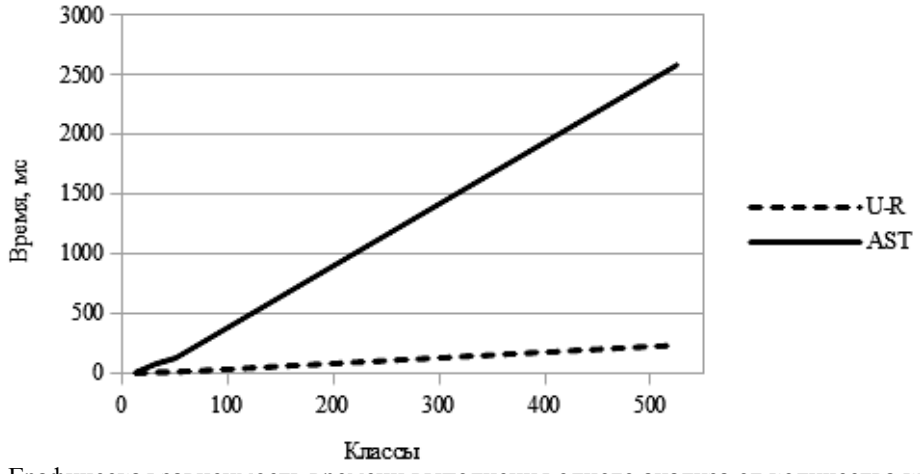

Рис. 1. Графическая зависимость времени выполнения одного анализа от количества классов

Выводы. Для выполнения статического анализа с использованием универсальных промежуточных представлений было выполнено математическое моделирование таких представлений. Модель показала преимущества, которые были предположены теоретически при разработке этих представлений. Были реализованы генераторы предложенных представлений, также были реализованы 2 варианта одного из анализаторов: на высокоуровневых представлениях и на абстрактном дереве разбора. На основе вариантов этого анализатора было выполнено численное моделирование для получения конкретной количественной оценки эффективности использования универсальных высокоуровневых представлений. Полученные количественные характеристики показывают, что в среднем реализация анализа таких представлений дает прирост производительности в 10 раз, что подтверждает теоретически рассчитанное преимущество, показанное математической моделью, при их использовании на практике.

Литература

1. Зубов М.В. Применение универсальных промежуточных представлений для статического анализа исходного программного кода / М.В. Зубов, А.Н. Пустыгин, Е.В. Старцев // Доклады ТУСУРа. – 2013. – №1(27).– С. 64–68.

2. Ахо А. Компиляторы: принципы, технологии и инструментарий / А. Ахо, М. Лам, Р. Сети, Д. Ульман. – М.: Вильямс, 2010. – 1184 с.

3. Серебряков В.А. Основы конструирования компиляторов / В.А. Серебряков, М.П. Галочкин. – М.: Эдиториал УРСС, 2001. – 224 с.

4. Зубов М.В. Сравнительный анализ существующих инструментов исследования программ по исходному коду / М.В. Зубов, А.Н. Пустыгин, Е.В. Старцев // Суперкомпьютерные технологии и открытое программное обеспечение. – Челябинск: Изд-во Челяб. гос. ун-та, 2013. – С. 37–44.

5. Зубов М.В. Краткий анализ и исследование помежуточных представлений исходного текста программ / М.В. Зубов, А.Н. Пустыгин, Е.В. Старцев // Суперкомпьютерные технологии и открытое программное обеспечение. – Челябинск: Изд-во Челяб. гос. ун-та, 2013. – С. 45–53.

6. Pylint – code analysis for Python [Электронный ресурс]. – Режим доступа: http://www.pylint.org/, свободный (дата обращения: 12.09.2014).

7. PMD [Электронный ресурс]. – Режим доступа: http://pmd.sourceforge.net/, свободный (дата обращения: 12.09.2014).

8. Linton M. Queries and Views of Programs Using a Relational Database System [Электронный ресурс]. – Режим доступа: www.eecs.berkeley.edu/Pubs/TechRpts/1983/5296.html, свободный (дата обращения: 12.09.2014).

9. Crew R. ASTLOG: A Language for Examing Abstract Syntax Trees / Proceedings of the Conference on Domain-Specific Languages. – Santa Barbara, California, 1997. – 15 p.

10. Moose technology [Электронный ресурс]. – Режим доступа: http://www.moosetechnology.org/, свободный (дата обращения: 12.09.2014).

11. Зубов М.В. Построение универсального представления графа потока управления для статического анализа исходного кода / М.В. Зубов, А.Н. Пустыгин, Е.В. Старцев // Девятая конференция «СПО в высшей школе»: тезисы докладов. – М.: Альт Линукс, 2014. – С. 46–51.

12. The GraphML File Format [Электронный ресурс]. – Режим доступа: http://graphml.graphdrawing.org/, свободный (дата обращения: 12.09.2014).

13. Control Flow – GNU Compiler Collection (GCC) Internals [Электронный ресурс]. – Режим доступа: http://graphml.graphdrawing.org/, свободный (дата обращения: 12.09.2014).

14. CSU tools for code analysis. GitHUB [Электронный ресурс]. – Режим доступа: https://github.com/exbluesbreaker/csu-code-analysis, свободный (дата обращения: 12.09.2014).

15. Фаулер М. Рефакторинг. Улучшение существующего кода. – СПб.: Символ-Плюс, 2003. – 268 с.

16. OpenJDK [Электронный ресурс]. – Режим доступа: http://openjdk.java.net/, свободный (дата обращения: 12.09.2014).

Зубов Максим Валерьевич
Аспирант каф. компьютерной безопасности и прикладной алгебры (КБиПА) ЧелГУ, г. Челябинск
Тел: +7-961-784-45-31
Эл. почта: zubovmv@gmail.com

Пустыгин Алексей Николаевич
Канд. техн. наук, доцент каф. КБиПА
Тел: +7-905-835-98-68
Эл. почта: p2008an@rambler.ru

Старцев Евгений Владимирович
Аспирант каф. КБиПА
Тел: +7-961-789-69-23
Эл. почта: slayer-gurgen@yandex.ru

Zubov M.V., Pustygin A.N., Startsev E.V.

Math modeling of universal multilevel intermediate representations for source code static analysis

This article describes using of source code universal multilevel intermediate representations for static analysis. Common math model of universal representation was developed to evaluate its efficiency and quality criterions. Models for universal class representation and universal control flow representation were introduced on the basis of common model. Numerical modeling of a analysis was made in practice to compute efficiency of using such representations.

Keywords: programming, static analysis, intermediate representation, math model, source code, refactoring, Java.