Оценка сложности алгоритмов построения дерева доминаторов в контексте имплементации алгоритмов анализа потока данных в реализации фронтенда компилятора языка программирования Solidity

В.Р. Столяров[1], С.В. Ларин[2], С.А. Скобельцын[1]

[1]Тульский государственный университет
[2]Тульский государственный педагогический университет

Аннотация: Данная статья рассматривает два из наиболее применимых алгоритмов построения дерева доминаторов в контексте статического анализа кода на языке программирования Solidity. Оба алгоритма: итеративный алгоритм Cooper, Harvey, Kennedy и алгоритм Lengauer-Tarjan считаются эффективными и широко используются на практике. В статье производится сравнение алгоритмов, оценка их сложности и выбор наиболее предпочтительного в контексте данного языка. Для сравнения были использованы такие критерии, как время выполнения и использование памяти. Итеративный алгоритм Cooper, Harvey, Kennedy показал более высокую производительность при работе с небольшими проектами, в то время как алгоритм Lengauer-Tarjan лучше справился с анализом более крупных проектов. Однако, в целом, итеративный алгоритм Cooper, Harvey, Kennedy оказался более предпочтительным в контексте Solidity, так как он показал более высокую эффективность и точность при анализе смарт-контрактов на данном языке программирования. В заключение, данная статья может быть полезна для разработчиков и исследователей, которые занимаются статическим анализом кода на языке Solidity, и могут использовать результаты и выводы данного исследования в своей работе.
Ключевые слова: дерево доминаторов, Solidity, сравнение алгоритмов.

Введение

Программное средство, осуществляющее статический анализ программ, трудно представить без механизма для анализа потока данных. Современный подход по решению проблемы анализа потока данных предполагает представление кода в SSA-форме [1]. Для того, чтобы её построить, требуется алгоритм построения дерева доминаторов. От выбора алгоритма зависит скорость построения SSA-формы и скорость анализа в целом. В данной статье мы рассмотрим существующие алгоритмы, попытаемся выбрать из них наиболее эффективные с точки зрения скорости работы и реализуем их в современном средстве анализа программного кода на языке Solidity [2] – одного из самых популярных языков

© Электронный научный журнал «Инженерный вестник Дона», 2007–2024

программирования для смарт-контрактов [3] на блокчейн платформе Ethereum [4, 5].

Статический анализ программ [6, 7] помогает автоматизировать поиск ошибок, которые часто допускают разработчики, в ходе написания кода. Для качественного анализа требуется также реализовать механизм анализа потока данных (Data Flow analysis [8]). Он необходим для того, чтобы знать значения переменных в тех или иных точках программы.

В данный момент в анализаторах и компиляторах применяются промежуточные представления кода [9], основанные на SSA-форме. Static Single Assignment form – это вид промежуточного представления, в котором каждой переменной присваивается значение только один раз. Такое представление упрощает работу с промежуточными представлениями кода, его анализ и дальнейшую оптимизацию.

Построение данной формы предполагает построение дерева доминаторов для графа потока управления [10]. В данной статье мы рассмотрим два алгоритма – классический и итеративный [11], приведем собственное описание итеративного алгоритма и сделаем выводы об их эффективности.

Описание алгоритмов

Итак, для начала дадим несколько определений.

Доминирование – это отношение одного узла a к другому узлу b в ориентированном графе, при котором при прохождении пути от начального узла до узла b путь всегда будет проходить через узел a.

Дерево доминаторов – это способ представления информации о доминировании. Каждому узлу графа соответствует некий набор узлов, которые являются его доминаторами. Однако для того, чтобы узнать всю информацию о доминаторах, достаточно знать только самого ближайшего

© Электронный научный журнал «Инженерный вестник Дона», 2007–2024

доминатора. Этот узел будет предком данного узла в дереве доминаторов. Таким образом, чтобы собрать полную информацию о доминаторах, следует пройти путь от нужного узла до корневого в дереве доминаторов.

Алгоритм Lengauer-Tarjan

Классический алгоритм построения дерева доминаторов, описанный Ленгауэром и Тарьяном, предполагает реализацию поиска ближайшего общего предка (LCA, Least Common Ancestor). Конечная сложность во многом зависит от реализации поиска. Предполагается использовать Link & Eval деревья [12]. На практике этот алгоритм часто работает за линейное время [13] и считается довольно эффективным.

Идея самого алгоритма состоит из нескольких этапов:

1) Выполняется обход графа в глубину (DFS), каждому узлу проставляется соотвествующее время входа (t_{in}).

2) Построение полудоминаторов. Полудоминатором вершины A называется такая вершина B, что $t_{in}(A) < t_{in}(B)$, причем существует путь из A в B.

3) На основе полудоминаторов строится конечное дерево доминаторов.

Опираясь на фундаментальную статью [13] можно сказать, что сложность алгоритма в критериях Big-O Notation:

1) $O(m * logn)$, где m – число рёбер графа, а n – это число вершин графа. В его классической версии

2) $O(m * a(m,n))$, где $a(m,n)$ – это обратная функция Акермана. Такая оценка корректна для улучшенной версии данного алгоритма [13].

© Электронный научный журнал «Инженерный вестник Дона», 2007–2024

Далее, при замерах производительности мы будем применять вторую версию данного алгоритма.

Итеративный алгоритм Cooper, Harvey, Kennedy

Итеративный алгоритм берет свое начало из следующего определения доминаторов:

$$DOM(n_0) = \{n_0\}$$
$$DOM(n) = \left(\bigcap_{p \in preds(n)} DOM(p)\right) \cup n$$

Каждый узел доминирует сам себя по определению. Доминатором начального узла является сам узел. Доминатор каждого последующего узла – это пересечение множества доминаторов всех предков этого узла, а также сам узел.

Простой итеративный алгоритм вычисления выглядит следующим образом:

для каждого узла, n

DOM[n] ←{1…N}

Изменён ←истина

Пока (Изменён)

Изменён ←ложь

для каждого узла, n, в порядке RLN:

$$\text{новый_набор} \leftarrow \left(\bigcap_{p \in preds(n)} DOM[p]\right) \cap \{n\}$$

если(новый_набор)≠DOM[n])

DOM[n] ←новый_набор

Изменён ←истина

© Электронный научный журнал «Инженерный вестник Дона», 2007–2024

Этот алгоритм не сложен для понимания и реализации. Однако, он непрактично медленный – версия Ленгауэра и Тарьяна работает быстрее до 900 раз [13].

Для улучшения производительности понадобится эффективная по потреблению памяти структура данных, которая способна быстро выполнять операцию пересечения. Сохранение порядка узлов в наборе может помочь в улучшении производительности. Если мы будем воспринимать набор как список, а объединение списков будет всегда производить вставку в конец, то в таком случае наборы доминаторов будут всегда иметь порядок узлов, в котором они добавлены. При таком порядке, если $DOM(a) \cap DOM(b) \neq \varnothing$, то конечный набор – это префикс обоих списков $DOM(a)$ и $DOM(b)$.

Это свойство позволяет реализовать пересечение как прямой проход через упорядоченные наборы, осуществляя попарное сравнение элементов. Если элементы соответствуют, то узел копируется в результирующий набор, и сравнение переходит к следующему элементу. Иначе, если элементы несоответствуют, или достигнут конец списка, пересечение останавливается и текущий узел добавляется как последний элемент своего набора доминаторов.

Для улучшения работы с памятью стоит отметить следующее свойство этих упорядоченных наборов. Для каждого узла, кроме узла n_0, имеет место следующее соотношение:

$$DOM(b) = \{b\} \cup \ IDOM(b) \cup \ IDOM\big(IDOM(b)\big) \ldots n_0$$

где $IDOM(b)$ – это ближайший доминатор узла b.

Такое соотношение позволяет реализовать структуру данных [14], называемую деревом доминаторов.

Набор $DOM(b)$ содержит узлы на пути через дерево доминатор от начального узла n_0 до узла b. Описанная операция пересечения создает набор $DOM(b)$ именно в таком порядке узлов на пути в дереве. Первый элемент $DOM(b)$ это n_0. Последний элемент $DOM(b)$ это b. Предпоследний элемент $DOM(b)$ – это $IDOM(b)$ – ближайший доминатор к узлу b. Поэтому, с такими упорядоченными наборами узлов, мы можем получать $IDOM(b)$ напрямую из узлов.

Отношение между DOM наборами и деревом доминаторов подсказывает альтернативную структуру данных. Вместо того, чтобы хранить отдельные наборы для каждого узла, алгоритм может представлять дерево доминаторов и считывать наборы из этого дерева. Алгоритм хранит один масств, $doms$, индексируемый узлом. Для узла b его включение в $DOM(b)$ представлено неявно. Запись $doms[b]$ содержит $IDOM(b)$. Запись $doms[doms[b]]$ содержит следующий элемент, т.е. $IDOM(IDOM(b))$. Обходя массив $doms$, начиная с b, мы можем построить путь через дерево доминаторов от b до n_0 и набор $DOM(b)$.

Чтобы использовать это представление, алгоритм должен производить пересечение начиная с конца DOM набора и двигаться к началу. Это обращает способ сравнения списков: проходя по ним, мы сравниваем элементы до тех пор, пока они не будут равны.

Ниже представлен псевдокод алгоритма вместе с операцией пересечения. Пересечение реализовано с помощью двух «указателей», где каждый указатель двигается независимо в зависимости от условий

© Электронный научный журнал «Инженерный вестник Дона», 2007–2024

сравнения. В нашем случае сравнение выполняются по номерам обратного обхода. Для каждого пересечения мы начинаем сравнение с конца, и, пока указатели не начнут указывать на одинаковый номер, мы будем сдвигать указатель, который указывает на меньший номер, назад на один элемент. Стоит помнить, что большие номера обратного обхода в дереве доминаторов будут соответствовать высшим узлам, именно поэтому мы передвигаем тот указатель, чей номер меньше другого. Когда два указателя начнут указывать на тот же элемент, функция вернет его. Набор, получившийся из пересечения, начинается с возвращенного элемента и идет до начального узла в массиве doms по цепочке.

```
    для каждого узла, b; инициализация массива доминаторов
    doms [b] ←Неопределено
doms [начальный узел] ←начальный узел
Изменён ←истина
пока (Изменён)
    Изменён ←ложь
    для каждого узла, n, в порядке RLN (кроме начального узла)
        новый_idom  ← первый (обработанный) предок b; нужно выбрать
        для каждого другого предка, p, узла b
            если (doms[p] ≠ Неопределено); т.е. доминатор уже
        вычислен
                новый_idom ←пересечение (p, новый_idom)
        если (новый_idom ≠ doms[b])
            doms[b] ← новый_idom
            Изменён ←истина
Функция пересечения (a, b)
    указатель1 ←a
    указатель2 ←b
```

© Электронный научный журнал «Инженерный вестник Дона», 2007–2024

```
пока (указатель1≠указатель2)
    пока (указатель1<указатель2)
        указатель1=doms[указатель1]
    пока (указатель2<указатель1)
        указатель2=doms[указатель2]
возврат указатель1
```

Сложность данного алгоритма в критериях Big-O Notation [15] – $O(N^2)$ для худших случаев.

Применение в Solidity

Несмотря на то, что оба предложенных алгоритма считаются эффективными – реальные программы содержат до 1000 узлов в графе потока управления – более простой алгоритм показывает себя лучше на графах меньшего размера [13].

Специфика языка такова, что подавляющее большинство кода содержит небольшие по размеру функции. Также этот алгоритм менее требовательный к работе с памятью чем классический алгоритм за счет эффективной организации используемой структуры данных. Этот алгоритм проще в реализации, что сокращает затраты на разработку компилятора и статического анализатора, а также повышает уверенность в корректности работы.

Рассмотрим в качестве примера следующую функцию, вычисляющую квадратный корень числа:

```
function _sqrt(uint y) private pure returns (uint z)
{
        if (y > 3) {
            z = y;
            uint x = y / 2 + 1;
            while (x < z) {
```

```
                z = x;
                x = (y / x + x) / 2;
            }
        } else if (y != 0) {
            z = 1;
        }
}
```

Граф потока управления для нее выглядит следующим образом (рис.1):

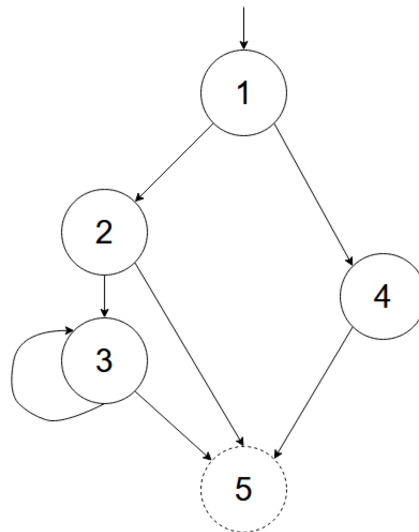

Рис. 1 – Вычисление квадратного корня

Применяя данный алгоритм, мы получим следующее дерево доминаторов (рис.2):

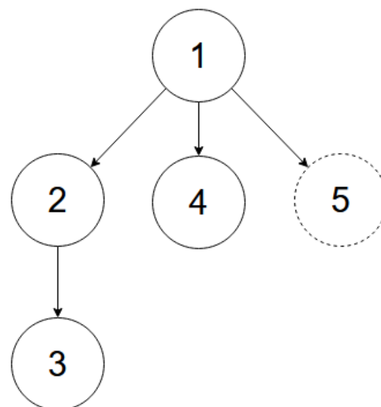

Рис. 2 – Дерево доминаторов

© Электронный научный журнал «Инженерный вестник Дона», 2007–2024

Сравнение производительности алгоритмов

Итак, мы имеем два алгоритма: итеративный алгоритм Cooper, Harvey, Kennedy и алгоритм Lengauer-Tarjan. Для выбора оптимального было проведено тестирование. Тестирование производилось путем построения дерева доминаторов. Содержание тестовой базы представляло следующий набор данных:

1) Файлы с ресурса solidity by example с примерами для новичков [16];

2) Реальные библиотеки для разработки смарт-контрактов (OpenZeppelin) [17];

3) Известные смарт-контракты, работающие в реальной сети (Tether) [18];

4) Вручную написанные тесты.

Во время работы отслеживался размер графа потока управления у функции и замерялось время построения графа. В результате имеем следующую таблицу, содержащую количество узлов графа к времени работы алгоритмов. Здесь N – количество узлов графа, t_{a1} – время работы алгорима Cooper, Harvey, Kennedy, t_{a2} – время работы алгоритма Lengauer-Tarjan:

Таблица №1

Сравнение производительности алгоритмов

N	t_{a1}	t_{a2}
>501	2,754	9,745
401-500	1,318	4,892
301-400	0,601	2,635
201-300	0,487	1,228
101-200	0,321	0,910
51-100	0,112	0,297
0-50	0,026	0,096

© Электронный научный журнал «Инженерный вестник Дона», 2007–2024

Выводы

В данной статье проведен анализ двух наиболее распространенных алгоритмов построения дерева доминаторов — итеративного алгоритма Cooper, Harvey, Kennedy и алгоритма Lengauer-Tarjan. Как видно из сравнения производительности, итеративный алгоритм Cooper, Harvey, Kennedy демонстрирует значительно более низкое время выполнения по сравнению с алгоритмом Lengauer-Tarjan при всех размерах графа потока управления. Это делает его более эффективным выбором для анализа смарт-контрактов на языке Solidity. Результаты данного исследования могут быть полезны исследователям в области алгоритмов для статического анализа и комиляторов, предоставляя рекомендации по выбору наиболее эффективного алгоритма для конкретных задач в экосистеме Solidity.

Литература

1. Sreedhar V.C. Translating out of static single assignment form // Static Analysis: 6th International Symposium, SAS’99 Venice, Italy, September 22–24, 1999 Proceedings 6. Springer Berlin Heidelberg, 1999. pp. 194-210.

2. Tikhomirov S., Voskresenskaya E., Ivanitskiy I., Takhaviev R., Marchenko E., Alexandrov Ya. Smartcheck: Static analysis of ethereum smart contracts // Proceedings of the 1st international workshop on emerging trends in software engineering for blockchain. 2018. pp. 9-16.

3. Dwivedi V., Norta A., Wulf A., Leiding B., Saxena S., Udokwu C. A Formal Specification Smart-Contract Language for Legally Binding Decentralized Autonomous Organizations // IEEE access. 2021. V. 9. pp. 76069-76082.

4. Oliva G.A., Hassan A.E., Jiang Z.M. An exploratory study of smart contracts in the Ethereum blockchain platform // Empirical Software Engineering. 2020. V. 25. pp. 1864-1904.

© Электронный научный журнал «Инженерный вестник Дона», 2007–2024

5. Гречко А.С., Кудрявцев О.Е. Калибровка умеренно устойчивых моделей Леви по данным криптовалют Bitcoin и Ethereum // Инженерный вестник Дона, 2019, №8 URL: ivdon.ru/ru/magazine/archive/N8y2019/6113.

6. Воротникова Т.Ю. Надежный код: статический анализ программного кода как средство повышения надежности программного обеспечения информационных систем // Информационные технологии в УИС. 2020. №2. С. 22-27.

7. Гинис Л.А., Вовк С.П. Определение четко доминирующих тактик для выработки альтернативных управляющих решений в условиях полной неопределенности // Инженерный вестник Дона, 2014, №2 URL: ivdon.ru/ru/magazine/archive/n2y2014/2327.

8. Khedker U., Sanyal A., Sathe B. Data flow analysis: theory and practice. CRC Press, 2017. 395 p.

9. Click C., Paleczny M. A simple graph-based intermediate representation // ACM Sigplan Notices. 1995. V. 30. №3. pp. 35-49.

10. Cytron R., Ferrante J., Rosen B.K., Wegman M.N., Zadeck F.K. Efficiently computing static single assignment form and the control dependence graph // ACM Transactions on Programming Languages and Systems (TOPLAS). 1991. V. 13. №4. pp. 451-490.

11. Быкова В.В. Математические методы анализа рекурсивных алгоритмов // Журнал Сибирского федерального университета. Математика и физика. 2008. Т. 1. № 3. С. 236-246.

12. Lengauer T., Tarjan R.E. A fast algorithm for finding dominators in a flowgraph // ACM Transactions on Programming Languages and Systems (TOPLAS). 1979. V. 1. №1. pp. 121-141.

13. Cooper K.D., Harvey T.J., Kennedy K. A Simple, Fast Dominance Algorithm // Software Practice & Experience. 2001. V. 4. №1-10. pp. 1-8.

© Электронный научный журнал «Инженерный вестник Дона», 2007–2024

14. Мельников Г.А., Губарев В.В. Итеративный метод построения деревьев регрессии // Вестник СибГУТИ. 2016. № 4(36). С. 59-67.

15. Edmonds J., Karp R. M. Theoretical improvements in algorithmic efficiency for network flow problems //Combinatorial Optimization—Eureka, You Shrink! Papers Dedicated to Jack Edmonds 5th International Workshop Aussois, France, March 5–9, 2001. Berlin, Heidelberg : Springer Berlin Heidelberg, 2003. pp. 31-33.

16. Solidity by Example // URL: solidity-by-example.org/ (дата обращения: 04.10.2024).

17. Liu H., Sun Y. Using My Functions Should Follow My Checks: Understanding and Detecting Insecure OpenZeppelin Code in Smart Contracts // 33rd USENIX Security Symposium (USENIX Security 24). USENIX Association, 2024. pp. 3585-3601.

18. Pierro G.A. Smart-graph: Graphical representations for smart contract on the ethereum blockchain // 2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, 2021. pp. 708-714.

References

1. Sreedhar V.C. Static Analysis: 6th International Symposium, SAS'99 Venice, Italy, September 22–24, 1999 Proceedings 6. Springer Berlin Heidelberg, 1999. pp. 194-210.

2. Tikhomirov S., Voskresenskaya E., Ivanitskiy I., Takhaviev R., Marchenko E., Alexandrov Ya. Proceedings of the 1st international workshop on emerging trends in software engineering for blockchain. 2018. pp. 9-16.

3. Dwivedi V., Norta A., Wulf A., Leiding B., Saxena S., Udokwu C. IEEE access. 2021. V. 9. pp. 76069-76082.

4. Oliva G.A., Hassan A.E., Jiang Z.M. Empirical Software Engineering. 2020. V. 25. pp. 1864-1904.

© Электронный научный журнал «Инженерный вестник Дона», 2007–2024

5. Grechko A.S., Kudryavtsev O.E. Inzhenernyj vestnik Dona, 2019, №8. URL: ivdon.ru/ru/magazine/archive/N8y2019/6113.

6. Vorotnikova T.Yu. Informatsionnye tekhnologii v UIS. 2020. №2. pp. 22-27.

7. Ginis L.A., Vovk S.P. Inzhenernyj vestnik Dona, 2014, №2. URL: ivdon.ru/ru/magazine/archive/n2y2014/2327.

8. Khedker U., Sanyal A., Sathe B. Data flow analysis: theory and practice. CRC Press, 2017. 395 p.

9. Click C., Paleczny M. ACM Sigplan Notices. 1995. V. 30. №3. pp. 35-49.

10. Cytron R., Ferrante J., Rosen B.K., Wegman M.N., Zadeck F.K. ACM Transactions on Programming Languages and Systems (TOPLAS). 1991. V. 13. №4. pp. 451-490.

11. Bykova V.V. Zhurnal Sibirskogo federal'nogo universiteta. Matematika i fizika. 2008. V. 1. № 3. pp. 236-246.

12. Lengauer T., Tarjan R.E. ACM Transactions on Programming Languages and Systems (TOPLAS). 1979. V. 1. №1. pp. 121-141.

13. Cooper K.D., Harvey T.J., Kennedy K. Software Practice & Experience. 2001. V. 4. №1-10. pp. 1-8.

14. Mel'nikov G.A., Gubarev V.V. Vestnik SibGUTI. 2016. № 4(36). pp. 59-67.

15. Edmonds J., Karp R. M. Combinatorial Optimization—Eureka, You Shrink! Papers Dedicated to Jack Edmonds 5th International Workshop Aussois, France, March 5–9, 2001. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003. pp. 31-33.

16. Solidity by Example. URL: solidity-by-example.org/ (accessed: 04.10.2024).

17. Liu H., Sun Y. 33rd USENIX Security Symposium (USENIX Security 24). USENIX Association, 2024. pp. 3585-3601.

© Электронный научный журнал «Инженерный вестник Дона», 2007–2024

18. Pierro G.A. 2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, 2021. pp. 708-714.

Дата поступления: 6.09.2024

Дата публикации: 17.10.2024

© Электронный научный журнал «Инженерный вестник Дона», 2007–2024