Санкт-Петербургский государственный университет

Математико-механический факультет
Кафедра системного программирования

Ножкин Илья Игоревич

Платформа для реализации межпроцедурных статических анализов на основе контекстно-свободной достижимости

Курсовая работа

Научный руководитель:
к. ф.-м. н., ст. преп. Григорьев С. В.

Санкт-Петербург
2019

Оглавление

Введение

Статический анализ — один из основных методов проверки корректности написанного кода. Современные IDE предоставляют обширные наборы готовых инспекций для поиска самых распространенных проблем в реальном времени. Однако, большинство из них ориентированы в первую очередь на внутрипроцедурный анализ и призваны в первую очередь улучшить общее качество кода и уберечь программиста от случайных ошибок. Межпроцедурные же взаимодействия, как правило, остаются на совести разработчика. Заметим также, что ввиду базовых принципов разработки, в частности, декомпозиции, именно межпроцедурные взаимодействия задают логику работы программы. Вследствие чего приемлемый результат оценки корректности программы, как полного комплекса, можно получить, только сохраняя и учитывая как можно больше связей между отдельными ее сущностями.

Вследствие этого, возникает необходимость в реализации межпроцедурных анализов, применимых в реальном времени, что, однако, является нетривиальной задачей, поскольку анализы такого рода характеризуются использованием всех известных фактов о программе, которые, в свою очередь нужно поддерживать в актуальном состоянии. Кроме того, крайне полезной была бы возможность иметь простые, специфичные для данной области инструменты для описания правил анализа. Отсюда в итоге следует идея создания инструмента, извлекающего из программы данные, удобные для проведения анализов, и одновременно предоставляющего абстракции, упрощающие их написание. О создании такого инструмента и пойдет речь в данной работе.

1. Постановка задачи

Основная цель работы — реализовать инструмент, позволяющий на основе информации, предоставляемой IDE, сформировать образ программы, и провести на его основе статический анализ посредством симуляции магазинных автоматов.

Работа естественным образом разделяется на 2 подзадачи: разработка расширения IDE, извлекающего первичную информацию и реализация непосредственно анализатора. В данной работе рассматривается вторая подзадача.

Выбранная задача также подразделяется на несколько необходимых элементов.

- Предварительно требуется провести обзор предметной области и существующих решений.
- Далее, необходимо разработать систему, учитывающую их недостатки и реализующую дополнительный интересующий функционал. Данный пункт разделяется на следующие подзадачи:
  - Реализовать систему агрегации данных, получаемых от IDE.
  - Обеспечить возможность симуляции исполнителя с магазинной памятью на построенной программе.
  - Используя полученные абстракции создать один или несколько анализов и обеспечить синхронизацию их результатов с IDE.
- Для проверки полученного результата необходимо провести апробацию инструмента на реальных проектах.

2. Обзор

2.1. Используемые технологии

Идея применения менее общих, чем прямое исполнение, методов для проверки программ достаточно широко исследуется. Самый близкий к описанной в данной работе концепции подход — сведение задачи межпроцедурного анализа к проверке контекстно-свободной достижимости в графе (Context-Free-Language reachability, CFL-r) [6]. Он заключается в двухэтапной обработке исходного кода. На первой стадии извлекается граф потока управления (Control-Flow Graph), ребра которого представляют собой интересующие операции. Например, вызов, возврат из процедуры, присваивание и другие. После чего в графе производится поиск путей, таких что конкатенация меток на ребрах, входящих в удовлетворяющий путь, содержится в языке, порождаемом заданной контекстно-свободной грамматикой. В силу равенства множества языков, порождаемых контекстно-свободными грамматиками и множества языков, принимаемых магазинными автоматами, данный подход эквивалентен предложенному в текущей работе.

2.2. Существующие инструменты

Из наиболее завершенных продуктов, соотносящихся с предметной областью, стоит обратить внимание на Graspan [2] — систему для операций над большими графами, ориентированную на проведение межпроцедурных анализов. Разработчики предоставляют инструмент, принимающий на вход граф и контекстно-свободную грамматику в нормальной форме Хомского, позволяя в дальнейшем обрабатывать вершины, между которыми существует путь, удовлетворяющий данной грамматике. Например, есть возможность построить замыкание графа вдоль всех путей, удовлетворяющих данной грамматике. Несмотря на крайне близкий к текущей работе подход, Graspan обладает одним существенным недостатком — ориентированностью на описание анализов грамматиками. Теоретическое равенство множеств языков, задаваемых КС-

грамматиками и магазинными автоматами, не всегда позволяет записать в виде грамматики логику, примитивно реализуемую в виде автомата. Кроме того, ввиду иммутабельности грамматик и их символьной природы, крайне нетривиально реализовать реакцию анализатора на данные, динамически получаемые во время выполнения.

Еще одно решение, являющееся базой для написания анализов — IncA [8]. Оно основано на поиске в графе некоторых шаблонов, для определения которых инструмент предоставляет специальный язык. Кроме того, важной особенностью является поддержка инкрементального обновления базы и результатов в ответ на изменения в исходном коде, что предоставляет возможность использовать данный продукт для проведения инспекций в реальном времени.

2.3. Разработки научной группы

Вопросы применения CFL-r в анализе программ также исследуются в рамках лаборатории языковых инструментов JetBrains. В одной из работ [3] используется алгоритм синтаксического анализа RNGLR, обобщенный для анализа графов. Предоставляется возможность построения всех возможных деревьев разбора для регулярного выражения, являющегося аппроксимацией генератора динамических программ, что в дальнейшем находит применение в реинжиниринге программного обеспечения.

Несмотря на первоначально иную цель упомянутой работы, разработанный в рамках нее инструмент предоставляет возможности, аналогичные Graspan, не ограничивая при этом структуру грамматики нормальной формой Хомского. В связи с этим была предпринята попытка использовать его в качестве основы для достижения цели, поставленной в текущей работе. Существенной проблемой, однако, снова стала невозможность выразить сложную динамическую логику в виде грамматики.

3. Решение

3.1. Архитектура

Центральной и базовой цастью полученного инструмента является симулятор магазинных автоматов.

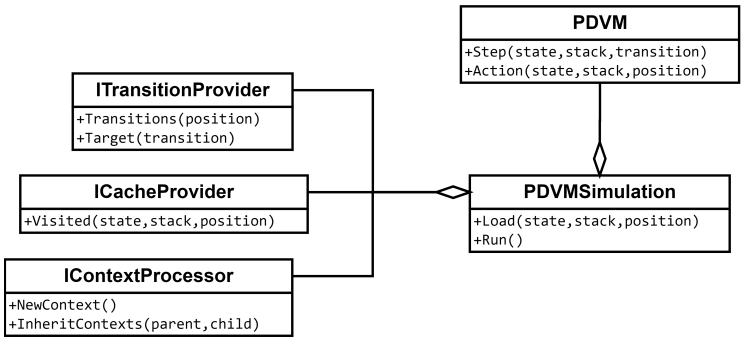

Рис. 1: Интерфейсы симулятора

Будучи спроектированным наиболее абстрактно, он позволяет сформулировать анализ в любых подходящих под задачу терминах. Для этого необходимо реализовать представленные на диаграмме 1 интерфейсы и запрограммировать автомат. Иными словами, требуется предоставить реализацию 3-х необходимых для симуляции сущностей: представление графа (ITransitionProvider), которое поставляет информацию о вершинах и исходящих ребрах; механизм кэширования (ICacheProvider), который поддерживает в актуальном состоянии множество обработанных дескрипторов; Обработчик контекстов (дескрипторов), который отвечает за порождение новых контекстов и за их наследование в соответствии с пользовательской семантикой. Кроме того, необходимо реализовать непосредственно автомат, имеющий возможность произвести либо шаг симуляции с переходом по ребру (Step), либо действие без перехода (Action). Единственным ограничением на рабочие множества является следующее требование: на позициях обрабатываемого графа, на множестве состояний автомата и множестве символов стека должно быть задано отношение равенства. Это требование вытекает из природы симуляции, которая во многом опирается на динамическое программирование.

Стоит обратить внимание также на метод программирования автомата. Разумеется, для обеспечения корректности симуляции решение автомата о дальнейшем шаге должно приниматься исключительно на основе 4-х параметров: текущей позиции, входного символа, состояния автомата и символа на вершине стека. Однако, ввиду потенциально большой мощности каждого из множеств, предоставляющих выше приведенные элементы, построение таблицы в явном виде может быть крайне нецелесообразным. Также, сущестует множество объектов, находящихся во взаимно-однозначном соответствии с элементами этих множеств, но не входящих в них явно и зачастую использование этих объектов может быть крайне полезным, но не нарушающим корректность. Поэтому, в данной реализации предлагается задание автомата в виде виртуальной машины, которая имеет несколько заранее заданных операций, таких как произвольные переходы в обрабатываемом графе, смена состояния и действия со стеком, но кроме того, обладает возможностью обращаться к любым внешним данным.

Вторым по важности элементом является система построения и модификации абстрактной программы, являющейся образом исходного кода. Она выполняет несколько важных функций.

Во-первых, компонует полученные от IDE отдельные классы в общий граф с возможностью прямой адресации для обеспечения переходов из одного метода в другой. Также она обеспечивает инкрементальное обновление образа, что позволяет в каждый момент времени поддерживать актуальное состояние данных и таким образом удовлетворяет условию, необходимому для использования анализатора в реальном времени.

Во-вторых, понижает избыточность данных, индексируя все содержащиеся в программе сущности, что также ускоряет операции поиска и сравнения.

Непосредственно исполняемый код каждого метода представляет собой небольшой граф, ребра которого являются операциями с известной семантикой и наследуются от некоторого базового класса. Стоит обратить внимание, что в процессе компоновки графы отдельных ме-

тодов не связываются между собой, а межпроцедурные переходы симулируются посредством переходов интерпретатора между методами с сохранением точки возврата на стеке.

Обе описанные сущности аггрегируются и используются сервисом, поддерживающим связь с плагином для IDE посредством сетевого соединения. Таким образом, инструмент в фоновом режиме извлекает, подготавливает и сохраняет необходимые для проведения анализа данные, и по запросу запускает симуляцию виртуальной машины, исполняющей скомпонованную программу и сигнализирующую о найденных ошибках.

3.2. Реализация

3.2.1. Симуляция

Основная идея симуляции — использование кэширования во избежание повторения одинаковых действий. Это создает необходимость в компактном и допускающем кэширование представлении множества стеков. Используемый в данной работе алгоритм предполагает применение стека, представленного в виде графа, (GSS — Graph Structured Stack [9]) для решения проблемы.

Таким образом, для выполнения анализа необходимо просимулировать работу магазинного автомата для всех содержащихся в графе путей так, чтобы для каждого пути существовала корректно обработанная последовательность состояний автомата и однозначно сопоставляемый ей стек из GSS.

Предлагаемый алгоритм предполагает вместо последовательного обхода графа в глубину с сопутствующей симуляцией автомата использовать обработку мгновенных состояний симуляции, которые являются независимыми и могут быть рассмотрены в любом порядке. Мгновенное состояние симуляции (далее — дескриптор) описывается тремя элементами: состоянием автомата, вершиной в GSS и позицией в анализируемом графе. Алгоритм выполняется в 2 стадии. На первом этапе все дескрипторы начальных состояний помещаются в очередь. Далее,

из очереди извлекается по одному дескриптору и для каждого ребра, исходящего из вершины в графе, сопоставленной дескриптору, выполняется один шаг симуляции автомата. Процесс продолжается до опустения очереди дескрипторов.

Рассмотрим подробнее шаг симуляции. На обработку автомату предоставляются следующие элементы: текущее состояние, символ на вершине стека (соответствующий вершине GSS, сопотавленной дескриптору), метка на обрабатываемом ребре. Ответ автомата состоит из информации о смене состояния и/или действии над стеком. Каждое из возможных действий над стеком предполагает отдельный метод обработки.

- **Добавление элемента в стек.** В процессе добавления потенциально формируется дескриптор, которому соспоставлена вершина GSS, ссылающаяся на текущую вершину. Рассмотрим два возможных случая.
  - **а.** Множество обработанных дескрипторов уже содержит один или несколько экземпляров, имеющих новое состояние, новый символ на вершине стека и новую позицию. Тогда добавляется ребро в GSS, направленное из вершины, соответствующей новому символу, к вершине, соответствующей текущему обрабатываемому дескриптору. Также, для обнаруженной вершины GSS повторно выполяются все операции снятия со стека, произошедшие с её участием.
  - **б.** Иначе. Порождается новая вершина в GSS, сформированный дескриптор добавляется в очередь на обработку.
- **Выталкивание верхнего элемента из стека.** В процессе выталкивания потенциально формируется один или более дескрипторов, каждому из которых сопоставлена вершина GSS, на которую ссылалась вершина, сопоставленная текущему дескриптору. Рассмотрим для каждого из них два возможных случая.

  - **а.** Дескриптор уже содержится в множестве обработанных дескрипторов. Тогда его обработка считается завершенной.
  - **б.** Иначе. Дескриптор добавляется в очередь на обработку.

- **Сохранение состояния стека.** Формируется новый дескриптор, содержащий новое состояние и новую позицию, но ссылающийся на текущую вершину GSS. Метод его обработки полностью совпадает с предыдущим пунктом.

Предоставляется возможность каждому из рассмотренных случаев сопоставить пользовательский обработчик события, который должен интерпретировать произошедшее в соответствии с пользовательской семантикой.

3.2.2. База данных

Компоновщик основывается на библиотеке, предоставляющей набор сущностей для удобного представления графов и операций над ними, QuickGraph [4]. Выбор обоснован тем, что библиотека хорошо зарекомендовала себя при разработке других приложений и она отвечала требованиям по обеспечению быстрого и стабильного исполнения операций над большими графами, хранящимися непосредственно в оперативной памяти, и предоставляла средства отладочного вывода. Архитектура компоновщика, впрочем, позволяет в будущем без труда перейти к любой графовой базе данных при наличии необходимости. Предметно-специфичная логика реализована вручную.

Протокол обмена с плагином предельно прост и предусматривает передачу сущностей, представляющих объекты исходной программы напрямую, посредством сериализации в XML или JSON. Сохранение накопленной базы данных на диск и последующее ее восстановление реализуется также с помощью сериалиализации в XML или JSON. Такой подход был выбран ввиду тривиальности реализации, однако он также негативно влияет на производительность и объем хранимых данных. В дальнейшем планируется перейти к использованию Google Protocol Buffers [1].

3.2.3. Интеграция и реализованные анализы

В качестве пробной IDE для интеграции выбрана Visual Studio с расширением ReSharper [5]. В качестве первого тестового анализа — отслеживание путей и источников данных для каждой переменной.

Задача формализуется, в частности, таким образом. Назовем неизменяемые, содержащие код элементы программы первичными сущностями. К ним относятся, например, классы как типы, их методы, анонимные функции. Все элементы, которые так или иначе могут ссылаться на первичные сущности назовем вторичными сущностями. К ним относятся локальные переменные и поля классов. Требуется для каждой вторичной сущности как можно точнее определить, на какое множество первичных сущностей она может ссылаться.

Польза от такого анализа заключается, например, в повышении точности при обработке вызовов методов при последующих анализах или в возможности проверить источники данных для каждой переменной.

Выбранный анализ, на первый взгляд, вполне реализуется предоставляемыми инструментами путем явной симуляции семантики передачи данных, например, следующим образом. Состоянием виртуальной машины принимается отслеживаемая в данный момент переменная (в корневых контекстах — некоторая виртуальная сущность, не связанная ни с одной переменной исходного кода). Символы на стеке — вызовы методов с сохранением точки возврата и информации о возвращаемых параметрах. При обнаружении операции, которая присваивает одну сущность другой, производится сравнение текущего состояния с правой частью присваивания, если они совпадают, происходит смена состояния на переменную, находящуюся в левой части.

Однако, стоит заметить, что есть множество неочевидных проблем. Самая существенная из них — вызовы методов объектов абстрактных классов или по интефейсу, заключается в том, что в момент вызова необходимо уже иметь информацию о всех возможных конкретных типах объекта, метод которого вызывается. Очевидно, что примитивный метод решения — перезапуск анализа до насыщения всех множеств

первичных сущностей, соответствующих каждой переменной, является слишком медленным для практического применения. Поэтому применяется техника, в реальном времени строящая транзитивное замыкание графа присваиваний переменных, одновременно перезапуская анализ инкрементально для всех фактов о возможных ссылках, добавленных при обновлении графа. Это еще раз показывает, что предоставляемый интерпретатор предоставляет обширные средства для расширения его возможностей.

Второй реализованный анализ является частным случаем Taint Tracking-a, а именно поиском путей передачи зараженных переменных к критически важным местам их использования, не проходящих через фильтры, гарантирующие безопасность данных. В виду того, что в отличие от предыдущей задачи, данная полностью формулируется исключительно в терминах поиска путей, нет необходимости для ее реализации использовать дополнительные построения, такие как, например, графы присваиваний переменных. Таким образом, реализация снова основывается на отслеживании передачи данных между переменными через присваивания, вызовы процедур и возвраты из них и одновременным отслеживанием прохождения текущей отслеживаемой переменной через фильтры. Так как отслеживание взаимодействий с полями классов с сохранением правильного их порядка является нетривиальным моментом, то следует также принять дополнительное правило, согласно которому, запись зараженной переменной в поле объекта приводит к заражению объекта целиком, после чего отслеживание продолжается для данного объекта целиком.

Важным минусом текущей реализации анализа является необходимость ручной разметки ролей методов (источник заражения, фильтр, сток) конечным пользователем, что, в частности, препятствует тестированию на существующих объемных проектах.

3.3. Апробация

Первый анализ был протестирован на решении компилятора с открытым исходным кодом Roslyn [7]. Ввиду отсутствия поддержки Visual Basic, часть связей в коде оказалась утерянной, из-за чего анализ не может быть точным.

В остальном, было успешно скомпоновано 12533 C# классов, содержащихся в 10290 файлах. Запуск осуществлялся на системе с Windows 10, Intel Core i7-7700HQ 2.8 ГГц, 16 Гб оперативной памяти.

Время анализа (без учета создания или загрузки базы данных) — в среднем 16 с.

Размер потребляемой памяти — до 1 Гб в процессе построения и до 3 Гб на время анализа.

Время построения базы данных несёт мало информации о разработанном решении, так как оно практически полностью зависит от скорости обработки кода на стороне IDE.

Корректность анализа, к сожалению, удалось проверить только эмпирическим путем. Большинство выбранных случайным образом переменных имели корректное множество возможных сущностей, у остальных результат анализа являлся подмножеством корректного множества. Таким образом, не было обнаружено случаев, при которых анализ сигнализирует о невозможных для данной переменной сущностях, однако иногда их множество было неполным.

Заключение

В рамках данной курсовой работы:

- Проведен обзор предметной области и существующих решений, выделено несколько недостатков, предложены их решения и дополнительные улучшения.
- Реализован прототип системы, решающий поставленные задачи, в том числе:
  - Реализована в виде удаленного сервиса система обработки и хранения данных, получаемых от IDE.
  - Предоставлены интерфейсы для расширения базы доступных анализов посредством построения виртуальных машин с магазинной памятью. На сервисной стороне также обеспечена поддержка их запуска и дальнейшего извлечения результатов.
  - Работа системы протестирована на двух анализах:
    Taint Tracking-е и анализе, собирающем информацию о возможных динамически формируемых связях между сущностями.
- Работа анализатора протестирована на проекте компиляторной платформы Roslyn.

Дальнейшие планы включают в себя в первую очередь оптимизации, более плотную интеграцию с IDE и повышение удобства использования инструментов для написания собственных анализов. Важным направлением является также реализация возможности инкрементального анализа, поддерживающего не только добавление кода, но и его изменение и удаление, так как время работы в среднем линейно возрастает в зависимости от размера исходного кода и с некоторого момента полный перезапуск анализа становится нерентабельным для применения в качестве инспекции реального времени.

Список литературы

[1] Google Protocol Buffers.

[2] Graspan: A Single-machine Disk-based Graph System for Interprocedural Static Analyses of Large-scale Systems Code / Kai Wang, Aftab Hussain, Zhiqiang Zuo et al. // SIGOPS Oper. Syst. Rev. — 2017. — . — Vol. 51, no. 2. — P. 389–404. — URL: http://doi.acm.org/10.1145/3093315.3037744.

[3] Grigorev Semen, Kirilenko Iakov. GLR-based Abstract Parsing // Proceedings of the 9th Central & Eastern European Software Engineering Conference in Russia. — CEE-SECR '13. — New York, NY, USA : ACM, 2013. — P. 5:1–5:9. — URL: http://doi.acm.org/10.1145/2556610.2556616.

[4] QuickGraph. — URL: https://archive.codeplex.com/?p=quickgraph.

[5] ReSharper. — URL: https://www.jetbrains.com/resharper/.

[6] Reps Thomas. Program Analysis via Graph Reachability // Proceedings of the 1997 International Symposium on Logic Programming. — ILPS '97. — Cambridge, MA, USA : MIT Press, 1997. — P. 5–19. — URL: http://dl.acm.org/citation.cfm?id=271338.271343.

[7] Roslyn. — URL: https://github.com/dotnet/roslyn.

[8] Szabó Tamás, Erdweg Sebastian, Voelter Markus. IncA: A DSL for the Definition of Incremental Program Analyses // Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering. — ASE 2016. — New York, NY, USA : ACM, 2016. — P. 320–331. — URL: http://doi.acm.org/10.1145/2970276.2970298.

[9] Tomita Masaru. Graph-structured Stack and Natural Language Parsing // Proceedings of the 26th Annual Meeting on Association

for Computational Linguistics. — ACL '88. — Stroudsburg, PA, USA : Association for Computational Linguistics, 1988. — P. 249–257. — URL: https://doi.org/10.3115/982023.982054.