DOI: 10.25728/avtprom.2021.06.12

Генерация кода для тестирования компиляторов с использованием генеративно-состязательных сетей

В.А. Петухов (Университет ИТМО)

Рассматривается задача генерации кода на произвольном языке программирования, анализируются существующие достижения в этой области, их достоинства и недостатки. Выделяется ряд проблем, которые остаются актуальными, в частности, перекладывание на алгоритмы машинного обучения правил языка (синтаксиса и некоторых правил семантики) вместо их формального описания в архитектуре модели. Для решения обозначенных проблем предлагается применить алгоритмы генеративно-состязательных сетей.

Ключевые слова: машинное обучение, генерация кода, генеративно-состязательные сети, тестирование компиляторов.

Введение

При разработке различных программных продуктов обычно используется инфраструктурное ПО – язык программирования (код), компилятор языка программирования и среда разработки. Инфраструктурное ПО вносит вклад в конечный продукт, в частности, в его качество [1]. Так, ошибка в компиляторе может привести к тому, что семантика скомпилированного кода не будет в точности соответствовать семантике кода на этом языке программирования (то есть в целевом коде будет содержаться ошибка), а это, в свою очередь, может привести к сбоям во время работы целевой программы [2]. При этом без компиляторов не разрабатывается практически ни один программный продукт, в связи с чем проверка их качества является актуальной задачей.

Тестирование компиляторов с целью проверки их качества представляет собой технически непростую задачу, поскольку число тестовых сценариев (фрагментов кода) для компилятора крайне велико [3]: их практически невозможно полностью перебрать автоматически, тем более крайне сложно покрыть какую-то существенную часть функциональности компилятора руками инженера-тестировщика. Отсюда вытекает задача генерации кода для тестирования компилятора при некоторых ограничениях [4]: сгенерированный код должен тестировать наиболее нетривиальные части компилятора, в которых с большей вероятностью могут содержаться ошибки.

Для решения поставленной задачи предлагается использовать синтез алгоритмов машинного обучения и некоторой формальной модели синтаксиса и семантики языка программирования, чтобы встроенной формальной модели поручить соблюдение хорошо формализуемых правил при генерации кода, а алгоритмам машинного обучения поручить использование сложно формализуемых правил.

Тестирование компиляторов

Компилятор, как и любая другая программа, может быть протестирован теми же, универсальными методами, – составлением вариантов использования и их проверкой на каждой новой версии. Но такой способ для компиляторов не является исключительно эффективным: число вариантов использования – фрагментов кода – крайне велико; даже на покрытие незначительной части функциональности компилятора понадобиться много времени. С другой стороны, если фрагменты кода для тестирования компилятора составляет человек, близко знакомый с возможностями соответствующего языка программирования и примерным пониманием степени стабильности этих возможностей, то результат может быть эффективен и позволит получить набор тестов, в которых в какой-то степени проверяются фрагменты кода, наиболее подверженные ошибкам. Составить такой набор тестов какому-либо автоматизированному генератору кода было бы более проблемно, так как по умолчанию генератор кода ничего не знает о том, какие части языка нуждаются в тестировании больше других (какие части более подвержены ошибкам); но охват по числу полученных возможных сценариев получился бы значительно больше: за время написания человеком простого фрагмента кода на несколько строк генератор смог бы сгенерировать сотни фрагментов кода и проверить на них компилятор.

Третьим возможным подходом к тестированию компилятора является компиляция наиболее больших и сложных пользовательских проектов (проектов, код которых написан на интересующем нас языке программирования (ЯП)). Преимущество такого подхода состоит, во-первых, в близости к конечной цели: как правило, при разработке и поддержке компилятора ЯП одной из главных целей является

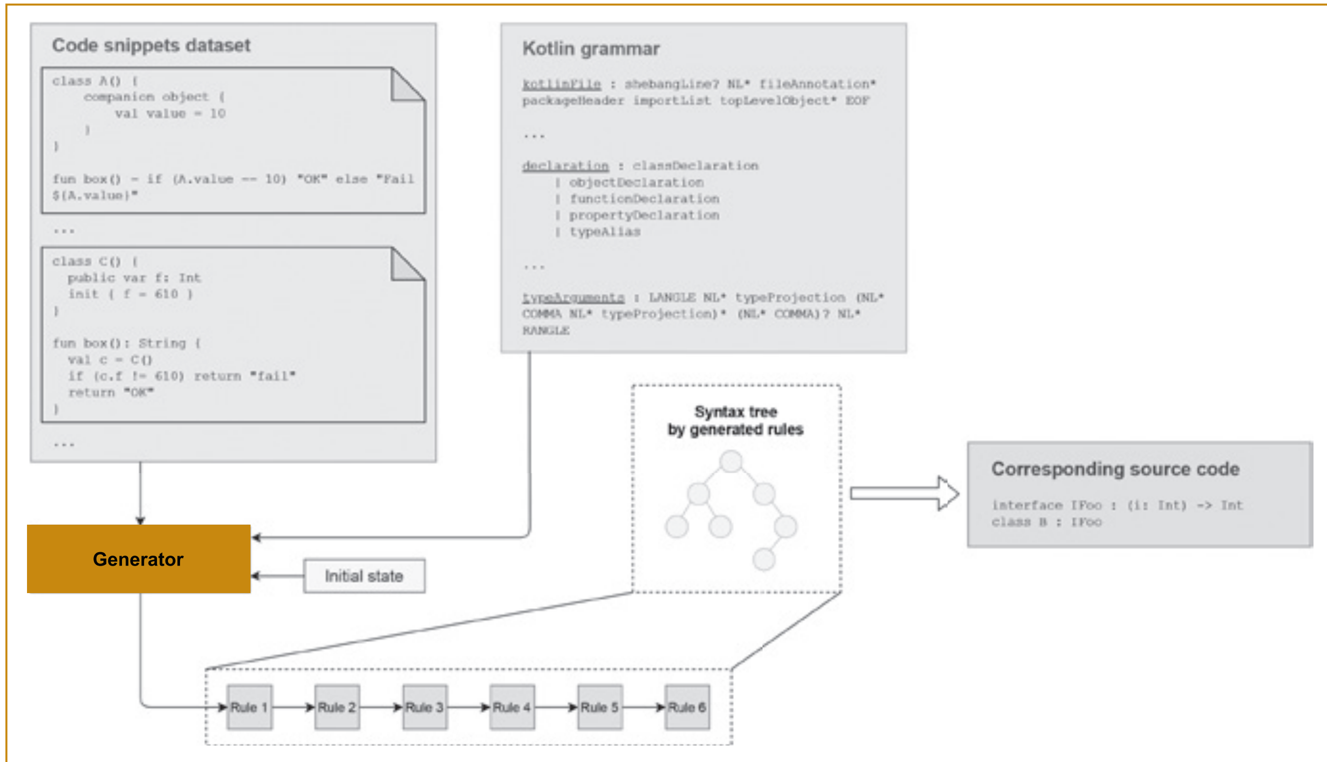

Рис. 1. Архитектура генеративно-состязательной сети для генерации программного кода

обеспечение корректной компилируемости реальных пользовательских проектов на этом ЯП. Эта цель обычно является более приоритетной [5], чем обеспечение правильной компилируемости какого-либо экзотического кода, который крайне редко встречается в реальных проектах, но который может быть в равной вероятности сгенерирован некоторым генератором кода или написан, например, разработчиком компилятора. Во-вторых, такой подход отличается относительной легкостью интеграции в процесс тестирования компилятора [6]: обычно достаточно склонировать репозиторий соответствующего проекта и настроить его сборку либо по какому-либо расписанию (например, раз в сутки), либо на каждое изменений в коде компилятора, либо перед очередным релизом. Таким образом, будет получен большой монолитный тест для компилятора, который ранее уже был написан разработчиками соответствующего проекта. Однако такой подход не применим для свежих возможностей языка: если та или иная функциональность еще не вошла в релиз компилятора, то почти наверняка в коде пользовательского проекта эта функциональность не используется, так как почти все проекты используют релизные версии компиляторов и библиотек. Если функциональность уже вошла в релиз, то вероятность ее использования так же невелика: необходимо время для обновления кодовой базы проекта.

Таким образом, наиболее эффективным подходом к тестированию компилятора будет совмещение всех трех подходов [7]: каждый из них сможет занять свою нишу в области тестирования компилятора и нивелировать недостатки другого подхода.

С ручным написанием компиляторных тестов и компиляцией больших пользовательских проектов проблем обычно не возникает. Автоматическая же генерация кода для тестирования на нем компилятора является хорошим полем для исследований, поскольку тестировщики стремятся сгенерировать как можно больше семантически корректного кода (чтобы затронуть самые дальние участки кода компилятора, не спровоцировать раннее завершение с пользовательской ошибкой про некорректный входной код). При этом для подавляющего большинства языков промышленного масштаба не существует полной формальной модели семантики, по которой можно было бы генерировать гарантированно правильный код. Соответственно появляется задача генерации такого кода, который бы удовлетворял не описанной формально семантике.

Применяемые подходы в генерации кода

В области генерации кода уже было проведено множество различных исследований и получены различные результаты. Существующие генераторы кода можно разделить на следующие виды:

[1] Backend — один из двух ключевых компонентов компилятора, который получает на вход абстрактное синтаксическое дерево (AST), хранящее логическую модель файла исходного кода; создает на выходе машинный код (не исполняемый файл, а всего лишь объектный, такой как *.obj или *.o).

```
Class A {
    fun  b() =
         null ?: "${c}"
}
```

Рис. 2. Пример проблемного сгенерированного кода, приводящего к утечке памяти в компиляторе

- вручную написанные генераторы для некоторого подмножества языка;
- генераторы, использующие синтаксическую модель языка (грамматику);
- генератора на основе алгоритмов машинного обучения.

Так, в [8] предлагается генерировать код по формально описанному генератору, который использует ограниченный набор конструкций языка. В [9] предлагается генерировать код, осуществляя случайный спуск по грамматическим правилам, сопоставляя им код на соответствующем ЯП. Авторы [10] применили алгоритмы машинного обучения для решения задачи генерации кода на языке OpenCL: они обучили генеративную модель структуре кода и нашли 67 ошибок в различных компиляторах.

Отметим, что в каждом из предложенных подходов есть как достоинства, так и недостатки: например, в случае вручную написанного генератора кода семантическая корректность гарантируется, но охват конструкций языка довольно мал (разработка генератора для всевозможных конструкций языка сопоставима по сложности с написанием компилятора). В случае генератора по грамматике охват максимальный (используется весь доступный синтаксис), сложность невысокая, но довольно мала доля генерируемого семантически корректного кода. При использовании же машинного обучения результат напрямую зависит от используемой модели и качества обучения, но сложность реализации модели и настройки гиперпараметров обычно довольно высока, так как требуется работа с высокоструктурированным данными – программным кодом.

Комбинированный подход

В данной работе предлагается использовать комбинированный подход, чтобы в какой-то степени нивелировать недостатки известных подходов – использовать для генерации кода генеративную модель, но заложить в её архитектуру работу с древовидной структурой – грамматикой и программным кодом, а именно: генерировать не исходных код непосредственно, а правила перехода по грамматике (https://techcrunch.com). Таким образом, генеративная сеть освобождается от необходимости обучения синтаксису языка, который заранее известен, и концентрируется только лишь на обучении наиболее сложным и не формализованным правилам языка – семантике.

Выбор целевого языка программирования

В качестве языка программирования, код для которого будет генерироваться, выбран Kotlin (https://kotlinlang.org/spec). Это относительно молодой язык с динамично развивающимися подсистемами (так, в Kotlin вер. 1.5 был полностью переписан back-end компилятор) и нуждающимися в тестировании. В 2017 г. Kotlin стал вторым официальным языком для разработки приложений под операционную систему Android (http://www.theverge.com), а в 2019 г. – предпочтительным (https://techcrunch.com).

Генеративно-состязательные сети

В качестве первой итерации апробации предложенного подхода предлагается использовать генеративно-состязательные сети [12], поскольку их архитектура может быть несложным образом модифицирована для генерации грамматических правил.

Генеративно-состязательная сеть - это методика обучения двух моделей:

1) генеративной G, которая по случайному шуму генерирует данные, подчиненные некоторому вероятностному распределению;

2) дискриминативной D, которая приписывает поступающим на ее вход данным, вероятность того, что они получены из тренировочного набора данных или сгенерированы моделью G.

Другими словами, генеративно-состязательная сеть состоит из двух нейронных сетей, одна из которых генерирует экземпляры G, а другая D пытается отличить правильные в каких-то терминах экземпляры от неправильных.

На рис. 1 изображена предлагаемая архитектура генеративно-состязательной сети, которая принимает на вход помимо набора данных с фрагментами кода (code snippets dataset) и начального состояния (initial state) ещё и грамматику (Kotlin grammar, https://kotlinlang.org). Это позволяет не обучаться заранее известному синтаксису языка, и таким образом генерировать не программный код напрямую, а правила перехода по грамматике, по которым уже можно сконструировать программный код.

Для завершения кодирования помимо генерации грамматических правил и соответствующего кода впоследствии необходимо также сгенерировать конкретные значения в некоторых листовых узлах (строки, числа, конкретные имена переменных, классов, функций и т. п.). В качестве первой итерации была реализована генерация случайных литеральных значений (строки, числа, символы и специальное значение null), а также генерация свободных на данный момент имен (во избежание некорректного переиспользования).

[2] https://youtrack.jetbrains.com/issue/KT-46455

Результаты экспериментов

С использованием описанного подхода был проведен ряд экспериментов и получены предварительные результаты. Генеративно-состязательная сеть с описанной архитектурой была обучена на наборе тестов компилятора. Такие тесты являются завершенным кодом, не имеющим дополнительных зависимостей, то есть нейронная сеть анализирует сразу «правильные» примеры, аналогичные которым хотелось бы получить и на выходе.

Помимо различного кода, не приводящего к ошибкам, был получен фрагмент кода, который в версии компилятора Kotlin 1.5.0 (https://kotlinlang.org) приводит к большому потреблению памяти и выбросу исключения OutOfMemoryError в последствии (рис. 2).

Хоть это было и известной компиляторной ошибкой на момент ее обнаружения генератором, сам факт нахождения действительной ошибки говорит о том, что генератор может использоваться в дальнейшем для нахождения других ошибок.

Заключение

В результате данного исследования была спроектирована модель генеративно-состязательной сети, работающая не с последовательностями, а с древовидными структурами данных, такими как программный код. Это позволило исключить избыточное обу-чение сети заранее известному синтаксису языка и нацелиться на обучение наиболее сложным не формализованным правилам языка – семантике.

В результатов первых экспериментов с помощью генератора получен фрагмент кода, на котором проявлялась уже известная ранее компиляторная ошибка, связанная с утечкой памяти во время компиляции. Такая находка доказывает, что сгенерированный код действительно способен обнаруживать компиляторные ошибки.

В перспективе планируется заложить в архитектуру сети не только синтаксис языка, но и некоторые легко формализуемые или уже формализованные элементы семантики, например, отношение подтипизации. Это позволит увеличить долю получаемого семантически корректного кода.

Список литературы

1. *Neelakantan, K., et al.* The Role of Compilers in Computer-System Performance // Current Science, vol. 60, no. 11, 1991, pp. 650–652
2. *Xuejun Yang, Yang Chen, Eric Eide, and John Regehr.* Finding and understanding bugs in C compilers // 32th ACM SIGPLAN Conference on Programming Language Design and Implementation. 2011. Pp. 283–294.
3. *Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang.* A Survey of Compiler Testing // ACM Comput. Surv. 53, 1, Article 4 (May 2020), 36 pages
4. *Andrew W. Appel.* Semantics-directed code generation // 12th ACM SIGACT-SIGPLAN symposium on Principles of programming languages. 1985. Pp. 315–324.
5. *Химонин Ю. И.* Сбор и анализ требований к программному продукту. http://mastefanov.com 6. A.S.Boujarwah, K. Saleh. Compiler test case generation methods: a survey and assessment // Information and Software Technology. Vol. 39. Is. 9. 1997. Pp. 617-625
6. *A.S. Boujarwah, K. Saleh.* Compiler test case generation methods: a survey and assessment // Information and Software Technology. Vol. 39. Is. 9. 1997. Pp. 617-625
7. *Chen, Junjie & Hu, Wenxiang & Hao, Dan & Xiong, Yingfei & Zhang, Hongyu & Zhang, Lu & Xie, Bing.* An empirical comparison of compiler testing techniques. 2016. pp. 180-190.
8. *Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon.* Semantic fuzzing with zest // 28th ACM SIGSOFT International Symposium on Software Testing and Analysis. 2019. Pp. 329–340
9. *Godefroid, Patrice & Kiezun, Adam & Levin, Michael.* Grammar-based Whitebox Fuzzing // Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). 2008. No 43. P. 206-215.
10. *Chris Cummins, Pavlos Petoumenos, Alastair Murray, and Hugh Leather.* Compiler fuzzing through deep learning // 27th ACM SIGSOFT International Symposium on Software Testing and Analysis. 2018. pp. 95–105
11. *X. Liu, X. Kong, L. Liu and K. Chiang.* TreeGAN: Syntax-Aware Sequence Generation with Generative Adversarial Networks // IEEE International Conference on Data Mining (ICDM). 2018. pp. 1140-1145

Петухов Виктор Алексеевич *– аспирант факультета «Информационных технологий и программирования»*
Университет ИТМО.
E-mail: i@victor.am

Роботы развивают автоматизацию в строительстве, делая его более безопасным и устойчивым

Девять из десяти строительных компаний прогнозируют кризис профессиональных навыков к 2030 г. При этом 81% из них заявляет, что внедрят роботов в ближайшие 10 лет, и считает, что требования к безопасности и охране окружающей среды ускоряют привлечение инвестиций в робототехнику. Согласно отраслевым прогнозам, общая стоимость мировой строительной индустрии вырастет к 2030 г. на 85% до 15,5 трлн долл. США.

Роботизация открывает огромные возможности в области повышения производительности, эффективности и производственной гибкости во всей строительной отрасли, включая автоматизацию изготовления модульных домов и строительных компонентов за пределами объекта строительства, роботизированную сварку и транспортировку материалов на строительных площадках, а также роботизированную 3D-печать домов и конструкций с учетом требований заказчика. Роботы не только делают отрасль более безопасной и рентабельной, но и повышают экологичность и снижают воздействие на окружающую среду за счет повышения качества строительства и сокращения строительных отходов.

Среди пилотных проектов роботизации от ABB, разработанных для повышения гибкости, производительности и качества, автоматизированное производство деревянных опор для крыш совместно с компанией Autovol в Канаде, роботизированная установка лифтов вместе с Schindler Lifts и роботизация производства сборных модульных домов в компании Intelligent City, которая позволила повысить эффективность производства на 15% и скорость на 38%, а также сократить объем отходов на 30%. Решение по роботизированной сварке Skanska повысило качество, производительность и безопасность сотрудников, автоматизировав производство стальных арматурных корзин на месте строительства. Технология позволила также снизить затраты и сократить воздействие на окружающую среду за счет того, что теперь не нужно перевозить громоздкие готовые арматурные корзины на строительные площадки.

Http:://www.abb.com