DOI: 10.15514/ISPRAS-2023-35(6)-6

Статический анализ на основе обобщённого абстрактного синтаксического дерева

[1,2] *В.О. Афанасьев*, ORCID: 0000-0002-8036-0633 <*vafanasiev@ispras.ru*>
[1] *А.Е. Бородин*, ORCID: 0000-0003-3183-9821 <*alexey.borodin@ispras.ru*>
[1,4] *К.И. Вихлянцев* <*vishkosty@ispras.ru*>
[1,3] *А.А. Белеванцев*, ORCID: 0000-0003-2817-0397 <*abel@ispras.ru*>

[1] *Институт системного программирования РАН,*
Россия, 109004, г. Москва, ул. А. Солженицына, д. 25.
[2] *Национальный Исследовательский Университет Высшая Школа Экономики,*
Россия, 101000, г. Москва, ул. Мясницкая, д. 20.
[3] *Московский государственный университет имени М.В. Ломоносова,*
Россия, 119991, Москва, Ленинские горы, д. 1.
[4] *Московский физико-технический институт,*
Россия, 141701, Московская область, г. Долгопрудный, Институтский переулок д.9.

Аннотация. В работе описывается универсальное представление для абстрактного синтаксического дерева (АСД), подходящее для статического анализа нескольких языков программирования. Предлагаемая схема анализа состоит в сохранении промежуточного представления в виде обобщенного АСД из компиляторов соответствующих языков и последующим анализом сохраненных деревьев. Мы реализовали такое представление для Java, Kotlin и Python. В анализаторе обобщенного АСД реализовано 27 детекторов. Мы описываем сущности предлагаемого представления, особенности его реализации для поддержанных языков. Приводим экспериментальные результаты скорости и качества анализа, а также сравнения анализа на обобщенном АСД с анализом, выполненным ранее на АСД конкретного компилятора. В итоге подход демонстрирует некоторое ухудшение скорости анализа, но позволяет разделить построение АСД для анализа и реализацию детекторов, что упрощает разработку АСД-детекторов в случае, когда количество поддерживаемых анализатором языков становится значительным.

Ключевые слова: статический анализ; поиск ошибок; АСД; синтаксический анализ; Java; Kotlin; Python; Svace.

Для цитирования: Афанасьев В.О., Бородин А.Е., Вихлянцев К.И., Белеванцев А.А. Статический анализ на основе обобщённого абстрактного синтаксического дерева. Труды ИСП РАН, том 35, вып. 6, 2023 г., стр. 103–120. DOI: 10.15514/ISPRAS–2023–35(6)–6.

Static Analysis Based on the Unified Abstract Syntax Tree

[1,2] *V.O. Afanasyev*, ORCID: 0000-0002-8036-0633 <*vafanasiev@ispras.ru*>
[1] *A.E.Borodin*, ORCID: 0000-0003-3183-9821 <*alexey.borodin@ispras.ru*>
[1,4] *K.I.Vihliantsev* <*vishkosty@ispras.ru*>
[1,3] *A.A. Belevantsev*, ORCID: 0000-0003-2817-0397 <*abel@ispras.ru*>

[1] *Institute for System Programming of the Russian Academy of Sciences,*
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.
[2] *National Research University Higher School of Economics,*
20 Myasnitskaya str., Moscow, 101000, Russia.
[3] *Lomonosov Moscow State University,*
Leninskie Gory, Moscow, 119991, Russia.
[4] *Moscow Institute of Physics and Technology,*
9 Institutskiy Pereulok, Dolgoprudny, Moscow Oblast, 141701, Russia.

Abstract. The paper describes a unified representation for an abstract syntax tree (AST) suitable for static analysis of several programming languages. The proposed analysis scheme consists of saving an intermediate representation in the form of a unified AST from compilers of the corresponding languages and subsequent analysis of the saved trees. We have implemented this described representation for Java, Kotlin and Python. The unified AST analyzer has 27 checkers. In the paper we present structure and entities of our unified AST, provide more details regarding language specifics that have to be reflected in the UAST representation. We give extensive experimental results that show UAST generation and analysis speed, analysis quality, and comparison with the old scheme of analyzing compiler ASTs where applicable. As a result, we see that we observe some degradation of analysis speed, but we pay it for the separation of AST construction and checkers' implementation. This separation allows easier support of many languages in the analyzer, where one can just generate UAST and support the required checker once within the UAST infrastructure instead of implementing a checker once per language.

Keywords: static analysis; search for defects; AST; parsing; Java; Kotlin; Python; Svace.

For citation: Afanasyev V.O., Borodin A.E., Vihliantsev K.I., Belevantsev A.A. Static analysis based on the unified abstract syntax tree. *Trudy ISP RAN/Proc. ISP RAS*, vol. 35, issue 6, 2023. pp. 103-120 (in Russian). DOI: 10.15514/ISPRAS-2023-35(6)-6.

1. Введение

В данной работе мы описываем универсальный статический анализатор, позволяющий единообразно описывать легковесные детекторы ошибок для множества языков программирования. Анализатор использует единое абстрактное синтаксическое дерево (АСД) для всех языков и в настоящее время поддерживает Java, Kotlin и Python. Реализация выполнена в рамках инструмента статического анализа Svace [1-3].

Анализ на основе АСД для каждого исходного файла получает на вход дерево, описывающее синтаксис анализируемого кода. Вершинами в этом дереве являются операторы языка, а листьями – соответствующие операнды.

Для анализа АСД в Svace использовался подход, при котором используется АСД компилятора соответствующего языка, и в рамках компилятора реализуются детекторы для поиска ошибок. Исключение – анализ АСД для языков C и C++, который выполняется на основе Clang Static Analyzer (CSA) [4]; с одной стороны, Clang Static Analyzer использует то же АСД, что и сам Clang, а с другой, помимо самого анализа АСД, в CSA можно создавать детекторы на основе статического символьного выполнения.

Основным недостатком схемы с реализацией детекторов внутри компиляторов является независимая кодовая база каждого детектора. Это, в свою очередь, приводит к необходимости реализовывать каждый детектор отдельно, невозможно создать общую

библиотеку для детекторов разных языков. Реализованные детекторы получаются неконсистентными и работают немного по-разному для разных языков. Как правило, детекторы в разных компиляторах реализуют разные люди с разным опытом. Это приводит к тому, что улучшение определенного детектора для одного языка не приводит к улучшению того же детектора для другого. Очевидным следствием является необходимость поддерживать больше кода и высокая стоимость разработки.

Общая идея анализатора на основе обобщённого АСД заключается в том, что в компиляторах остаётся только код для генерации АСД, которое имеет единый формат для всех языков. Все детекторы при этом реализованы в отдельном анализаторе.

Подобная схема работоспособна при определенных условиях:

- Типы узлов для абстрактных синтаксических деревьев поддерживаемых узлов должны пересекаться. В таком случае код обработки общих узлов разделяется между языками, а специфические для каждого языка узлы могут обрабатываться отдельно. При этом правомерно использовать разные компромиссы при создании анализатора: специфические узлы могут обрабатываться по-своему для полного учета особенностей языка, могут выражаться через общие для получения некоего «среднего» варианта обработки, а могут вообще пропускаться.
- Нужно учитывать потенциальное замедление анализа, т.к. компилятор должен сохранить АСД для последующего анализа.

В данной статье мы опишем, как был реализован наш анализатор обобщенного АСД, с какими проблемами мы столкнулись, сравним недостатки таких анализаторов с их преимуществами. Оценим целесообразность использования нашего подхода.

Для языков Java и Kotlin анализатор Svace уже имел реализации на основе АСД в компиляторах Javac и Kotlinc [5] соответственно. Мы сравним оба подхода.

Для языка Python у нас не было другого варианта анализа, и на примере этого языка мы оценим сложность добавления поддержки нового языка в анализатор на обобщённом абстрактном синтаксическом дереве (далее UAST).

2. Схема анализа с UAST

Общая схема UAST-анализа приведена на рис. 1 и содержит две фазы: трансляцию и анализ.

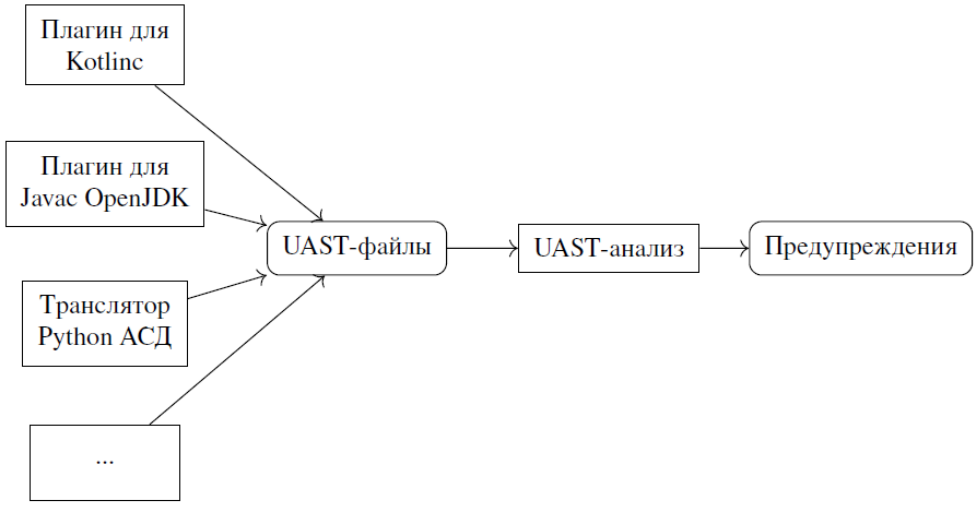

Рис. 1. Схема UAST-анализатора
Fig. 1. Scheme of the UAST analyser

Во время фазы трансляции для каждого языка запускаются трансляторы – программы, получающие на вход исходную программу и конвертирующие каждый файл с исходным кодом этой программы в файл в формате UAST. Помимо этого, трансляторы параллельно могут выполнять и другие действия, например, генерировать внутреннее представление для глубокого межпроцедурного анализа, выполняемого основным движком Svace.

Трансляторы для поддерживаемых нашим анализатором языков реализованы следующим образом:

- Трансляция AST языка Java осуществляется при помощи плагина компилятора Javac, использующего Compiler Tree API [6];
- Трансляция AST языка Kotlin осуществляется при помощи плагина компилятора Kotlinc, использующего PSI [7];
- Трансляция AST языка Python осуществляется при помощи скрипта на языке Python, использующего библиотеку ast [8].

Процесс анализа программы в Svace разделён на две фазы: контролируемую сборку [9] и анализ. На этапе сборки строится промежуточное представление программы, которое является входными данными для анализа. Такое разделение позволяет производить сборку и анализ на разных машинах, что удобно, т.к. к ним могут предъявляться разные требования. Для сборки требуется правильно настроить окружение. А для анализа может требоваться определённое количество оперативной памяти и желательно большее количество процессоров для его ускорения.

На фазе анализа происходит чтение промежуточных файлов в формате UAST, и для каждого такого файла происходит запуск ряда детекторов. То, какие детекторы будут запущены, зависит от языка оригинальной программы. В нашем анализаторе реализованы как детекторы, которые работают для всех языков (т.е. логика их работы не зависит от самого языка), так и специфичные для одного или нескольких поддерживаемых языков. Нами было реализовано 27 детекторов, реагирующих на следующие ситуации:

- Неявное создание кортежа из одного значения (Python);
- Плохой клон кода (Java, Kotlin, Python);
- Использование метода `wait` вне цикла, из-за чего программа уязвима к так называемому ложному пробуждению – spurious wakeup (Java, Kotlin);
- Ловимое исключение имеет тип `NullPointerException` (Java, Kotlin);
- Слишком общий тип ловимого исключения (Java, Kotlin);
- Отступ инструкции, следующей за условным оператором, создаёт впечатление вложенности, что вводит в заблуждение (Java, Kotlin);
- Пустой `catch`/`except`-блок (Java, Kotlin, Python);
- Пустой `synchronized`-блок (Java);
- Управление из ветви оператора `switch` может перейти к другой ветви (Java);
- Неправильные значения аргументов при создании диапазона (Kotlin);
- Результат операции не зависит от значений аргументов (Java, Kotlin, Python);
- Реализация метода `Iterator::hasNext` вызывает метод `Iterator::next` (Kotlin);
- Реализация метода `Iterator::next` не бросает исключение `NoSuchElementException` (Kotlin);
- Отсутствие `default`-ветви в операторах `switch` и `when` (Java, Kotlin);

- Использование изменяемого значения в качестве значения аргумента по умолчанию у функции (Python);
- Перекрытие имён (Java, Kotlin);
- Излишние или недостижимые проверки на тип в `when`-выражении (Kotlin);
- Использование оператора `return` в finally-блоке (Java, Kotlin, Python);
- Дублирующийся код в ветвлениях (Java, Kotlin, Python);
- Неправильный порядок аргументов при вызове функции (Java, Kotlin, Python);
- Неправильный порядок операций при использовании тернарного оператора (Java);
- Слишком общий тип бросаемого исключения (Java, Kotlin);
- Недостижимый `catch`-блок для `checked`-исключения (Java);
- Результат операции сравнения не используется (Kotlin);
- Использование неправильного значения в качестве аргумента оператора `synchrëonized` (Java);
- Сравнение по ссылке там, где ожидалось сравнение по значению (Java, Kotlin);
- Точка с запятой, стоящая после оператора `if`, что делает его бессмысленным (Java).

Все детекторы работают на уровне одного файла. То есть ошибки, для поиска которых требуется знание кода из разных файлов и/или модулей, в текущей реализации не обнаруживаются. Межмодульные ошибки также можно обнаруживать – для этого для каждого языка необходимо добавить этап связывания, который будет сопоставлять символы и типы из разных файлов и модулей. Данное ограничение было сделано намеренно, чтобы упростить реализацию.

В частности, подобный подход позволил реализовать кэш для анализа. В ходе такого анализа для каждого дерева строится хэш по содержимому. Этот хэш уникально описывает АСД. После анализа АСД результаты в виде предупреждений сохраняются в кэше, где хэш является ключом. Если при следующем анализе в кэше уже есть результаты для данного ключа, то анализ не запускается, а сразу возвращаются сохранённые результаты. Анализ с кэшем для случая, когда меняется небольшое количество файлов анализируемой программы, позволяет существенно ускорить результирующий анализ, т.к. реальный анализ будет выполняться только для изменившихся файлов.

3. Описание сущностей UAST-анализатора

UAST-анализ оперирует тремя основными сущностями: узлы АСД, символы и типы. Каждый узел:

- Хранит номер строки и столбца (позицию в коде) для первого и последнего символа, относящегося к узлу;
- Может иметь связанный тип (если это предусмотрено логикой данного узла). К примеру, с узлом идентификатора переменной будет связан тип данной переменной;
- Может иметь связанный символ (если это предусмотрено логикой данного узла). К примеру, узел идентификатора переменной будет иметь символ, указывающий на данную переменную;
- Может иметь дополнительные атрибуты, представленные другими узлами, типами, символами и прочими объектами (числами, строками, коллекциями и т.д.). К примеру, узел определения функции хранит в себе список узлов для всех инструкций данной функции, а узел константы хранит значение этой константы.

В нашем представлении используется 101 конкретный узел, из которых:

- 57 используются для двух или трёх языков;
- 9 используются только для Java;
- 13 используются только для Kotlin;
- 22 используются только для Python.

Помимо конкретных узлов, в нашем представлении имеется 17 общих узлов, от которых наследуются все прочие (к примеру, узел-«родитель» для всех выражений). Данные узлы позволяют детекторам абстрагироваться от специфики конкретного языка, реализуя только обработчики для общих надузлов.

Символы являются сущностями программы, которые можно отличить друг от друга по уникальному идентификатору (обычно имени[1]) и области видимости. Например:

- Локальные и глобальные переменные;
- Поля и свойства классов/структур/перечислений;
- Классы/структуры/перечисления;
- Параметры функций/методов/конструкторов;
- Функции/методы/конструкторы.

Под типами в UAST понимаются статические типы компилируемых языков – UAST-анализатор оперирует только теми типами, которые были известны в момент компиляции. Для интерпретируемых языков и языков с динамической типизацией информация о типах редко может быть определена статически, поэтому все типы считаются неизвестными, и информация о них никак не используется UAST-анализатором. Так происходит для языка Python (типы известны только для частных случаев вроде литералов, определений функций и определений пользовательских типов).

4. Особенности отдельных языков

4.1 Java

Как описывалось выше, транслятор для языка Java реализован в виде плагина компилятора Javac, использующего Compiler Tree API. Вся необходимая информация о сущностях предоставляется компилятором, и единственной задачей является лишь правильная трансляция этих сущностей в сущности UAST. В частности, несмотря на то, что UAST-анализ работает на уровне одного файла, информацию о типах и символах, определённых в других файлах, но используемых в текущих, удаётся сохранить в полном объёме.

Тем не менее, при реализации транслятора для языка Java мы столкнулись с рядом проблем.

Во-первых, некоторые сущности в АСД компилятора Java представлены одними и теми же узлами, из-за чего может возникать неоднозначность. К примеру, локальные переменные методов, поля классов и параметры функций представляются узлом `VariableTree`, а для определений классов, интерфейсов, аннотаций, перечислений и `record`-классов используется узел `ClassTree`. Подобного рода неоднозначности недопустимы в UAST – это сильно усложняет анализ, так как детекторам, реагирующим на конкретные типы узлов, приходится отличать подобные случаи самостоятельно. Поэтому такие сущности компилятора необходимо транслировать в разные сущности UAST, отличая их либо по наличию модификаторов (как в случае с `ClassTree`), либо по внешнему контексту (как в случае с `VariableTree`).

[1] Иногда уникальным идентификатором символа выступает некоторое целое число. Это важно для анонимных функций и классов, не имеющих имени, но для которых UAST-анализатор имеет символы и должен уметь их различать.

Второй проблемой при трансляции АСД языка Java является то, что АСД, предоставляемое посредством Compiler Tree API, не всегда соответствует исходному коду. В частности, мы столкнулись со следующими проблемами:

1. Для классов, в которых отсутствует конструктор, создаётся фиктивный конструктор без аргументов;
2. В классах, в которых отсутствует явный вызов `super-` или `this`-конструктора, добавляется неявный вызов `super`-конструктора без аргументов;
3. Элементы `record`-классов генерируются как поля внутри тела класса.

Решения этих проблем в нашем случае такие:

1. Фиктивные определения конструкторов отличаются от явных конструкторов по модификатору `GENERATEDCONSTR` у соответствующего символа компилятора. Такие определения не транслируются в обобщенное представление;
2. Вызовы методов, которые называются `this` или `super`, не имеют аргументов и имеют неизвестную позицию в исходном коде, считаются синтетическими и не транслируются в обобщенное представление;
3. `Record`-классы генерируются следующим образом. Ищется группа из последовательных полей класса, находящихся в самом верху соответствующего узла АСД, которые имеют моодификаторы `GENERATED_MEMBER`, `RECORD`, `PARAMETER` и не являются статическими. Эти поля считаются элементами `record`-класса, стоящими в его заголовке. Все прочие определения считаются определениями в теле `record`-класса.

Отметим также, что данный транслятор можно было бы реализовать не как плагин компилятора, а как надстройку над инструментом TreeSitter [10]. Мы не используем данный инструмент по ряду причин:

- TreeSitter предоставляет доступ только к АСД исходной программы. Информация о типах и символах данным инструментом не предоставляется, поэтому генерация данной информации должна быть возложена на транслятор. В нашем же подходе вся информация о типах и символах предоставляется самим компилятором и требует лишь трансляции в унифицированное представление;
- Svace во время фазы контролируемой сборки запускает компиляторы исходных языков, генерирующие промежуточное представление для основного анализа. Подключение плагина позволяет переиспользовать построенную компилятором информацию, требуемую UAST-анализатором, без каких-либо дополнительных расходов. При реализации транслятора на базе TreeSitter повторный парсинг и построение АСД занимали бы дополнительное время.

4.2 Kotlin

Как и транслятор для языка Java, транслятор для языка Kotlin реализован в качестве плагина к компилятору этого языка.

Одной из отличительных особенностей языка Kotlin является то, что почти все конструкции языка являются выражениями. Например, код приведённый в листинге 1, является полностью корректным. Операторы `continue` и `return` в языке Kotlin являются выражениями, благодаря чему они могут быть использованы как операнды в других операциях (в данном случае – при приведении типов и суммировании). Из-за подобных особенностей языка Kotlin, все конструкции, являющиеся выражениями, приходится считать выражениями и в UAST-анализаторе.

```
fun test() {
    for (i in (-5..5)) {
```

```
            println(i)
            val x = continue as Int +
                return as Int
            println(x)
        }
    }
```

Листинг 1. Использование continue и return в качестве выражений
Listing 1. Using continue and return as expressions

Язык Kotlin допускает перегрузку операторов. В частности, можно перегрузить операторы составного присваивания для встроенных типов, что допускает код из листинга 2. Это приводит к тому, что в UAST-анализаторе мы считаем, что слева от присваивания может стоять любое выражение даже в тех языках, которые такого не допускают[2].

```
fun test() {
    1 += 1
}
operator fun Int.plusAssign(value: Int) {
    println(this + value)
}
```

Листинг 2. Перегрузка оператора составного присваивания в Kotlin
Listing 2. Augmented assignment operator overloading in Kotlin

4.3 Python

Для понимания особенностей, с которыми пришлось столкнуться при реализации поддержки UAST для Python, необходимо рассмотреть, как исходный код программы на Python компилируется в байт-код. В качестве компилятора мы выбрали CPython – наиболее распространённую, эталонную реализацию языка программирования Python.

CPython компилирует исходный код в байт-код в несколько шагов [11]:

1. Разбиение исходного кода на токены;
2. Преобразование последовательности токенов в АСД;
3. Трансформация АСД в последовательность команд;
4. Построение графа потока управления и его оптимизация;
5. Генерация байт-кода на основе графа потока управления.

Построение таблицы символов компилятором CPython является промежуточным этапом при переходе со 2 на 3 шаг, к которому мы не можем получить доступа[3]. Поэтому UAST-представление строится на основе 2-го шага компиляции без сгенерированной компилятором CPython таблицы символов. Это приводит нас к необходимости создать и поддерживать собственную промежуточную таблицу символов при трансформации АСД в UAST-формат.

Для создания корректной промежуточной таблицы символов были учтены основные особенности языка Python [12]:

1. Области видимости переменных (блоки кода);

[2] Альтернативным решением этой проблемы может являться то, что в анализаторе будут иметься два узла для присваивания – общее, допускающее все выражения в левой части, и второе, допускающее в левой части только некоторые выражения. При этом второй узел будет наследником первого.

[3] Доступ к этой информации можно получить при наличии собственного модифицированного компилятора. Но поддержка своей версии компилятора – куда более трудоёмкая задача, чем представленная далее реализация.

2. Правила привязки переменных;
3. Специальные инструкции языка global [13] и nonlocal [14].

В языке Python существуют три области видимости: область видимости модуля (файла) или глобальная, область видимости класса и область видимости функции. Если объявить переменную в текущей области видимости, то она перекроет собой все переменные с таким же именем, которые расположены в областях видимости более высокого уровня.

Когда переменная используется в левой части выражения присваивания, она привязывается к текущей области видимости (см. листинг 3). В иных случаях сначала переменная будет искаться в текущей области видимости, а потом на верхних уровнях, заканчивая областью видимости модуля. Если ни в одной области видимости не будет найдено переменной с таким именем, то во время выполнения интерпретатором будет выброшено исключение.

```
a = 3
b = 7

def func():
    a = 13
    print(a) # 13
    b += 1 # NameError:
           #   name 'b' is not defined
```

Листинг 3. Правила привязки переменных
Listing 3. Name binding rules

Инструкции языка `global` и `nonlocal` позволяют изменить значение переменных, объявленных во внешних областях видимости. Инструкция `global` даёт указание интерпретатору искать переменную в области видимости модуля, в то время как инструкция `nonlocal` говорит о том, что переменная должна искаться во внешней области видимости.

```
 1| a = 3
 2|
 3| def func():
 4|     b = 7
 5|
 6|     def nested_func():
 7|         nonlocal b
 8|         b += 1
 9|
10|     nested_func()
11|
12|     global a
13|     a += b
14|
15| func()
16| print(a)  # 11
```

Листинг 4. Специальные инструкции языка global и nonlocal
Listing 4. Special global and nonlocal language statements

Инструкция nonlocal на 7 строчке листинга 4 даёт указание интерпретатору использовать в области видимости функции `nested_func` переменную с именем `b` из области видимости функции `func`, то есть переменную `b`, объявленную на 4 строчке. В то время как инструкция `global` указывает интерпретатору на необходимость искать переменную с именем `a` в

глобальной области видимости, поэтому в области видимости функции `func` будет использоваться переменная `a`, объявлена в самой первой строчке листинга.

Другой особенностью, с которой мы столкнулись, является то, что язык Python имеет динамическую типизацию переменных. Следовательно, в АСД отсутствует информация о типах. Для моделирования такой ситуации в UAST введён универсальный тип – UAnyType, который указывает на то, что тип значения может быть любым. Исключениями являются типы, которые связаны с объявлением функции или класса, а также константные выражения:

```
1| def func():
2|     ...
3|
4|
5| class Cls:
6|       ...
7|
8|
9| func = smth
```

Листинг 5. Типизация объявлений в UAST для Python
Listing 5. Typing of declarations in UAST for Python

То есть для примера из листинга 5 символ для идентификатора `func` на строке 1 будет иметь функциональный тип, а символ идентификатора `Cls` на строке 5 – тип класса. Однако при использовании переменной с идентичным именем в левой части присваивания на строке 9 транслятор создаст в динамической таблице символов новую переменную с именем `func` и типом `UAnyType`.

5. Результаты

5.1 Время анализа

При использовании UAST добавляется дополнительный этап в виде построения универсального промежуточного представления. Более того, в нашей реализации это представление сохраняется на диск. Поэтому можно ожидать замедление анализа.

В табл. 1 представлены данные времени сборки и анализа для проектов с открытым исходным кодом[4]. Для оценки Java анализа мы использовали исходный код ОС Android 11, для оценки Kotlin – исходный код компилятора Kotlinc версии 1.5.30, для Python – проект PyTorch версии 2.1.0.

Как можно заметить, замедление сборки на Java и Kotlin проектах очень незначительное – замедление для AOSP-11 составило около 3%, для Kotlinc-1.5.3 – около 2%. Сборка для PyTorch замедлилась примерно в три раза – скорее всего, это связано с тем, что транслятор для Python написан на языке Python.

[4] Сборка проектов проводилась на вычислительной машине с характеристиками: CPU: 16 cores 2.1GHz, RAM: 256Gb. Анализ проектов проводился на вычислительной машине с характеристиками: CPU: 8 cores 3.4GHz, RAM: 64Gb.

Табл.1. Результаты времени сборки и анализа
Table 1. Build and analysis time results

Проект	Язык	Размер, MLOC	Время сборки без UAST, мин	Время сборки с UAST, мин	Время анализа без UAST, мин	Время анализа с UAST, мин
ОС Android 11	Java	33	152	157	67	69
Kotlinc-1.5.30	Kotlin + Java	1.9 + 1.0	97	99	19	20
PyTorch-v2.1.0	Python	1	6	19	5	5

5.2 Сравнение результатов для Java и Kotlin

Мы вручную просмотрели результаты анализов для языков Java и Kotlin. Табл. 2 и 3 содержат сводные результаты нашей разметки. Для детекторов, которыми были выдано не более 50 предупреждений, были размечены все предупреждения. Для детекторов, которые выдали более 50 предупреждений, было размечено по 50 случайно выбранных предупреждений. В таблицах представлена информация только для детекторов, которыми было выдано хотя бы одно предупреждение. В среднем процент истинных предупреждений для Java и Kotlin составил более 90%.

Табл.2. Результаты разметки для Kotlinc-1.5.30
Table 2. Kotlinc-1.5.30 analysis results

Детектор	Количество истинных	Количество ложных	Процент истинных
Плохой клон кода	2	4	33%
Слишком общий тип ловимого исключения	50	0	100%
Пустой catch/except-блок	50	0	100%
Отсутствие default-ветви в операторах `switch` и `when`	42	0	100%
Управление из ветви оператора `switch` может перейти к другой ветви	0	1	0%
Реализация метода `Iterator::next` не бросает исключение `NoSuchElementException`	12	3	80%
Перекрытие имён	37	13	74%
Дублирующийся код в ветвлениях	19	0	100%
Неправильный порядок операций при использовании тернарного оператора	0	1	0%
Неправильный порядок аргументов при вызове функции	7	0	100%
Суммарно	**219**	**22**	**91%**

Практически все UAST-детекторы имеют качество не хуже, чем их неунифицированные версии. Из приведённых детекторов только два пострадали от унификации: детектор перепутанных местами аргументов и детектор плохих клонов кода.

Так как ранее в Svace использовался подход с анализом без унифицированного АСД, правильно было бы сравнить старую и новую реализации и определить, не вызвала ли унификация каких либо ухудшений. В табл. 4 и 5 представлены результаты сравнения детекторов для старой (без UAST) и новой (с UAST) реализаций. Сравнивались только

детекторы, которые выдали как минимум одно истинное предупреждение в старой реализации. В последнем столбце таблицы приведена доля истинных срабатываний старой реализации, которые были найдены и в новой.

Табл.3. Результаты разметки для ОС Android 11
Table 3. Android 11 analysis results

Детектор	Количество истинных	Количество ложных	Процент истинных
Плохой клон кода	40	10	80%
Использование метода `wait` вне цикла, из-за чего программа уязвима к т.н. spurious wakeup	40	5	89%
Слишком общий тип ловимого исключения	50	0	100%
Пустой catch/except-блок	50	0	100%
Ловимое исключение имеет тип `NullPointerException`	50	0	100%
Отступ инструкции, следующей за условным оператором, создаёт впечатление вложенности, что вводит в заблуждение	11	5	69%
Отсутствие default-ветви в операторах `switch` и `when`	50	0	100%
Пустой synchronized-блок	18	0	100%
Управление из ветви оператора switch может перейти к другой ветви	50	0	100%
Результат операции не зависит от значений аргументов	50	0	100%
Использование оператора return в finally-блоке	8	0	100%
Перекрытие имён	44	6	88%
Дублирующийся код в ветвлениях	49	1	98%
Неправильный порядок операций при использовании тернарного оператора	22	28	44%
Слишком общий тип бросаемого исключения	46	0	100%
Недостижимый catch-блок для checked-исключения	0	2	0%
Неправильный порядок аргументов при вызове функции	13	3	81%
Использование неправильного значения в качестве аргумента оператора synchronized	24	2	92%
Суммарно	**615**	**62**	**91%**

Качество первого детектора снизилось из-за того, что изначальная его версия, реализованная нами в компиляторе языка Java, работала на уровне нескольких файлов. Реализация же в UAST-анализаторе работает исключительно на уровне одного файла, соответственно, ошибочные случаи, в которых вызывается функция из другого файла, новой реализацией не обнаруживаются.

Качество детектора плохих клонов кода снизилось, так как изначально существовало две совершенно разных реализации этого детектора: одна для языка Java, другая – для Kotlin. В качестве эталонной реализации была выбрана версия для языка Java, и по большей части реализация в UAST-анализаторе основывается на ней. Соответственно, ухудшение данного детектора на Kotlin-проектах – ожидаемый эффект.

В листинге 6 приведён пример одного из срабатываний на ОС Android 11 (код отредактирован для компактности). В данном примере программист хотел проверить значение переменной `uri` на `null` и в зависимости от этого получить либо пустую строку, либо результат вызова метода `toString`. Но приоритет операций в языке Java отличается от того, что подразумевал автор – сначала будет выполнена операция конкатенации строк, затем сравнение со значением `null`, а только потом тернарный оператор. Таким образом, результат этой операции всегда будет равен `uri.toString()`.

Табл.4. Сравнение результатов с и без UAST-анализатора для ОС Android 11
Table 4. Comparison of analysis results with and without UAST analyser for Android 11

Детектор	Истинных без UAST	Истинных с UAST	Процент совпавших истинных
Плохой клон кода	40	40	100%
Использование метода `wait` вне цикла, из-за чего программа уязвима к т.н. spurious wakeup	40	40	100%
Слишком общий тип ловимого исключения	50	50	100%
Пустой catch/except-блок	50	50	100%
Ловимое исключение имеет тип `NullPointerException`	50	50	100%
Отступ инструкции, следующей за условным оператором, создаёт впечатление вложенности, что вводит в заблуждение	4	11	100%
Отсутствие default-ветви в операторах switch и when	50	50	100%
Пустой synchronized-блок	18	18	100%
Управление из ветви оператора switch может перейти к другой ветви	50	50	98% (49 из 50)
Результат операции не зависит от значений аргументов	49	50	100%
Использование оператора return в finally-блоке	8	8	100%
Дублирующийся код в ветвлениях	18	49	100%
Слишком общий тип бросаемого исключения	46	46	100%
Неправильный порядок аргументов при вызове функции	26	13	42% (11 из 26)
Использование неправильного значения в качестве аргумента оператора synchronized	23	24	96% (22 из 23)

Табл.5. Сравнение результатов с и без UAST-анализатора для Kotlinc-1.5.30
Table 5. Comparison of analysis results with and without UAST analyser for Kotlinc-1.5.30

Детектор	Истинных без UAST	Истинных с UAST	Процент совпавших истинных
Плохой клон кода	6	2	17% (1 из 6)
Слишком общий тип ловимого исключения	49	50	100%
Пустой catch/except-блок	50	50	100%
Отсутствие default-ветви в операторах switch и when	42	42	100%
Реализация метода `Iterator::next` не бросает исключение `NoSuchElementException`	12	12	100%

```
Log.w(TAG,
    "Exception for URI:" + uri == null ? "" : uri.toString(), e);
```

Листинг 6. Пример срабатывания анализатора на ОС Android 11
Listing 6. Example of emitted warning for Android 11

5.3 Анализ Python-проектов

Процесс добавления языка программирования Python в схему UAST-анализатора занял примерно 800 человеко-часов. Это время включает в себя написание модуля трансляции из Python АСД в UAST, дополнение общих для всех языков детекторов в соответствии с семантическими особенностями языка и добавление двух детекторов, работающих только для языка Python: детектора неявного создания кортежа из одного значения и детектора, находящего использования изменяемых значений в качестве значений аргумента по умолчанию. Суммарная кодовая база для поддержки языка Python составила примерно 4 тысячи строк кода, из которых 3 тысячи на Python и 1 тысяча на Java.

В табл. 6 приведены результаты разметки для проекта PyTorch. Общая доля истинных срабатываний составила 98%.

В листингах 7 и 8 представлены примеры срабатываний UAST-детекторов для Python (код отредактирован для компактности).

Табл.6. Результаты разметки для pytorch-v2.1.0
Table 6. Pytorch-v2.1.0 analysis results

Детектор	Количество истинных	Количество ложных	Процент истинных
Неявное создание кортежа из одного значения	50	0	100%
Плохой клон кода	14	4	78%
Пустой catch/except-блок	50	0	100%
Результат операции не зависит от значений аргументов	50	0	100%
Использование изменяемого значения в качестве значения аргумента по умолчанию у функции	4	0	100%
Использование оператора return в finally-блоке	2	0	100%
Дублирующийся код в ветвлениях	34	0	100%
Неправильный порядок аргументов при вызове функции	11	0	100%
Суммарно	**215**	**4**	**98%**

```
class Test:
  def __init__(self, lst=[1, 2, 3]):
    self.lst = lst


t1 = Test()
print(t1.lst)  # [1, 2, 3]
t1.lst.append(4)
print(t1.lst)  # [1, 2, 3, 4]

t2 = Test()
print(t2.lst)  # [1, 2, 3, 4]
```

Листинг 7. Пример срабатывания детектора, обнаруживающего изменяемые значения в качестве аргументов по умолчанию
Listing 7. Example of a warning emitted by the mutable default argument checker

```
primals = tree_map(
    lambda x: x if isinstance(x, torch.Tensor) else x, primals
)
```

Листинг 8. Пример срабатывания детектора дублирующихся ветвей
Listing 8. Example of a warning emitted by the similar branches checker

6. Похожие работы

Использование обобщенных абстрактных синтаксических деревьев является популярным подходом. Во многих случаях такой подход используется IDE для синтаксического анализа и рефакторинга. IDE X-Develop [15] имеет АСД для нескольких языков (C#, Java,Visual Basic, ASP, XML, JavaScript), которое может использоваться для рефакторинга программы.

Среда TexMo [16] имеет UAST, содержащий только общие лексические элементы, а отношения между строками описываются отдельными связями вида «ключ-значения». Среда не производит никакого существенного анализа, а лишь помогает отслеживать связи между элементами программы, в том числе на разных языках.

Компания JetBrains в своих IDE предоставляет фреймворк, называемый UAST [17]. Он является языконезависимой прослойкой над АСД конкретных языков, которая позволяет разрабатывать плагины, в том числе системы сборки, линтеры, статические анализаторы для языков, работающих поверх JVM. При помощи данного фреймворка упрощается реализация и поддержка данных инструментов, так как они реализуются лишь единожды сразу для нескольких языков. Тем не менее, интегрированный в среду разработки анализ имеет ограничения на сложность, так как должен выполняться в режиме реального времени и не мешать работе пользователя.

Инструмент DMS [18] предназначен для написания спецификаций и их реализаций на разных языках программирования, а также поддержания связей между ними и выполнения трансформаций кода. Программа DMS была применена к ряду коммерческих приложений на Java и C++, в которых выполнялся поиск клонов кода, упрощение директив препроцессора, генерация кода для микроконтроллеров.

Инструмент Bauhaus [19] для анализа и обратной инженерии программ представляет АСД поддерживаемых языков в виде графа, узлы которого формируют единую иерархию классов. Разработчики реализовали базовые консервативные анализы потока данных и управления поверх этого графа, включая анализ указателей, а также выполнили поиск взаимных блокировок и мертвого кода.

В [20] предлагается использовать представление на основе языка XML, содержащее основную информацию о классах программы и связях между ними. Представление генерируется парсерами соответствующих языков. В прототипе системы XML-представление генерируется для языков Java и Python и используется для анализаиерархии классов.

Другим подходом является построение UAST и предоставление декларативного языка для описания ошибочных ситуаций. В частности, в анализаторе Klocwork используется язык KAST [21, 22], расширяющий XQuery, для реализации АСД-детекторов.

Альтернативным решением является использование промежуточного представления, специализированного только для одного языка. Предыдущие версии анализатора Svace использовали такой подход. Хотя Svace имеет общее представление для анализируемых языков, это представление не включает в себя АСД. Поэтому каждый АСД-анализатор реализуется независимо для поддерживаемых языков программирования. В данной статье мы уже описали плюсы и минусы такого подхода.

Теоретически детекторы, реализованные на АСД от компилятора, имеют больше информации об исходной программе и могут работать быстрее. На практике при нехватке ресурсов на поддержку темп развития АСД-детекторов для разных языков был различным, некоторые редкие детекторы практически не развивались.

Инструмент Infer [23, 24] имеет промежуточное представление SIL, используемое при проведении анализа. Это представление не включает в себя синтаксис программы. Для реализации АСД-детекторов Infer используется декларативный язык для написания линтеров [25]. Этот язык, основанный на АСД компилятора Clang, ограничен языками, поддерживаемыми этим компилятором. Он недоступен для Java и C#, для которых возможен

анализ на основе SIL-а. Более того, он объявлен устаревшим (deprecated). Таким образом, отсутствие общего представления в данном случае помешало создать АСД-детекторы для Java и C#.

7. Заключение

Создание универсального статического анализатора оказывается возможным для довольно отличающихся языков программирования, таких как Kotlin и Python. Анализ с обобщенным представлением позволяет упростить разработку детекторов наоснове поиска шаблонов в АСД.

На наш взгляд, такой подход, имея незначительные ухудшения в скорости анализа, предпочтительнее, чем реализация отдельных анализаторов для каждого языка, т. к. позволяет разделить проблему построения АСД-представления и реализацию детекторов. Благодаря этому сложность реализации пропорциональна $N + M$, где N – количество поддерживаемых языков программирования, а M – количество детекторов. Для независимой реализации эта сложность пропорциональна их произведению $N \times M$, что становится существенным по мере роста количества поддерживаемых языков.

При наличии АСД-представления написание детекторов является относительно несложной проблемой. Тем не менее, на примере Infer видно, что эта работа не производится. Таким образом, использование общего представления является существенным для анализатора, поддерживающего несколько языков программирования.

Список литературы / References

[1]. В. П. Иванников, А. А. Белеванцев, А. Е. Бородин, В. Н. Игнатьев, Д. М. Журихин, А. И. Аветисян и М. И. Леонов. Статический анализатор Svace для поиска дефектов в исходном коде программ. Труды ИСП РАН , 26(1):231—250, 2014. DOI: 10.15514/ISPRAS-2014-26(1)-7.

[2]. A. Belevantsev, A. Borodin, I. Dudina, V. Ignatiev, A. Izbyshev, S. Polyakov, E. Velesevich and D. Zhurikhin. Design and development of Svace static analyzers. In 2018 Ivannikov Memorial Workshop (IVMEM), pp.3—9. IEEE, 2018.

[3]. A.E. Borodin and A. A. Belevantsev. A static analysis tool Svace as a collection of analyzers with various complexity levels. Proceedings of the Institute for System Programming of the RAS, 27(6):111—134, 2015.

[4]. Clang static analyzer. URL: https://clang-analyzer.llvm.org (дата обращения 02.10.2023).

[5]. V. Afanasyev, S. Polyakov, A. Borodin and A. Belevantsev. Kotlin from the point of view of static analysis developer. Programming and Computer Software, 49(7):549—558, 2023. DOI: 10.1134/S0361768823070022.

[6]. Compiler tree api. Английский. URL: https://docs.oracle.com/javase/8/docs/jdk/api/javac/tree/index.html (дата обращения 18.09.2023).

[7]. Intellij platform plugin sdk. Psi elements. Английский. URL: https://plugins.jetbrains.com/docs/intellij/psi-elements.html (дата обращения 18.09.2023).

[8]. Ast — abstract syntax trees. Английский. URL: https://docs.python.org/3/library/ast.html (дата обращения 18.09.2023).

[9]. A. Belevantsev, A. Izbyshev and D. Zhurikhin. Monitoring program builds for svace static analyzer. System administrator, (7-8):135—139, 2017.

[10]. Treesitter. Introduction. Английский. URL: https://tree-sitter.github.io/tree-sitter/ (дата обращения 18.12.2023).

[11]. Python developer’s guide. Compiler design. Английский. URL: https://devguide.python.org/internals/compiler/ (дата обращения 28.09.2023).

[12]. Execution model. Английский. URL: https://docs.python.org/3.8/reference/executionmodel.html#execution-model (дата обращения 13.10.2023).

[13]. The global statement. Английский. URL: https://docs.python.org/3.8/reference/simple_stmts.html#the-global-statement (дата обращения 27.10.2023).

[14]. The nonlocal statement. Английский. URL: https://docs.python.org/3.8/reference/simple_stmts.html#the-nonlocal-statement (дата обращения 27.10.2023).
[15]. D. Strein, H. Kratz и W. Lowe. Cross-language program analysis and refactoring. In 2006 Sixth IEEE International Workshop on Source Code Analysis and Manipulation, pp 207—216. IEEE, 2006.
[16]. R.-H. Pfeiffer и A. Wasowski. Texmo: a multi-language development environment. In European Conference on Modelling Foundations and Applications, pp 178—193. Springer, 2012.
[17]. Uast – unified abstract syntax tree. Intellij platform plugin sdk. Английский. URL: https://plugins.jetbrains.com/docs/intellij/uast.html (дата обращения 03.10.2023).
[18]. I. D. Baxter. Dms: program transformations for practical scalable software evolution. In Proceedings of the International Workshop on Principles of Software Evolution, pp 48—51, 2002.
[19]. A. Raza, G. Vogel и E. Plodereder. Bauhaus–a tool suite for program analysis and reverse engineering. In Reliable Software Technologies–Ada-Europe 2006: 11th Ada-Europe International Conference on Reliable Software Technologies, Porto, Portugal, June 5-9, 2006. Proceedings 11, pp 71—82. Springer, 2006.
[20]. М. В. Зубов, А. Н. Пустыгин и Е. В. Старцев. Применение универсальных промежуточных представлений для статического анализа исходного программного кода. Доклады Томского государственного университета систем управления и радиоэлектроники , (1 (27)):64—68, 2013.
[21]. С. В. Сыромятников. Декларативный интерфейс поиска дефектов по синтаксическим деревьям: язык kast. Труды Института системного программирования РАН , 20:51—68, 2011.
[22]. Н. Л. Луговской и С. В. Сыромятников. Применение языка kast для преобразования исходного кода и автоматического исправления дефектов. Труды Института системного программирования РАН , 25:51—66, 2013.
[23]. D. Harmim, V. Marcin и O. Pavela. Scalable static analysis using Facebook Infer. I, VI-B, 2019.
[24]. D. Harmim. Advanced static analysis of atomicity in concurrent programs through Facebook Infer. Proceedings of the Excel@ FIT'21.
[25]. Ast language (al). Английский. URL: https://fbinfer.com/docs/checker-linters/ (дата обращения 02.10.2023.

Информация об авторах / Information about authors

Виталий Олегович АФАНАСЬЕВ – магистрант факультета компьютерных наук НИУ ВШЭ, сотрудник ИСП РАН. Сфера научных интересов: компиляторные технологии, статический анализ, JVM языки.

Vitaly Olegovich AFANASYEV – master's student at the Faculty of Computer Science, NRU HSE, employee of ISP RAS. Research interests: compiler technologies, static analysis, JVM languages.

Алексей Евгеньевич БОРОДИН – кандидат физико-математических наук, старший научный сотрудник ИСП РАН. Сфера научных интересов: статический анализ исходного кода программ для поиска ошибок.

Alexey Evgenevich BORODIN – Cand. Sci. (Phys.-Math.) - researcher. Research interests: static analysis for finding errors in source code.

Константин Игоревич ВИХЛЯНЦЕВ – студент факультета ФРКТ Физтеха, сотрудник ИСП РАН. Сфера научных интересов: компиляторные технологии, статический анализ, язык Python.

Konstantin Igorevich VIHLIANCEV – PSRECT MPTI student, ISP RAS researcher. Research interests: compiler technologies, static analysis, Python.

Андрей Андреевич БЕЛЕВАНЦЕВ – доктор физико-математических наук, ведущий научный сотрудник ИСП РАН, профессор МГУ. Сфера научных интересов: статический анализ программ, оптимизация программ, параллельное программирование.

Andrey Andreevich BELEVANTSEV – Dr. Sci (Phys.-Math.), Leading Researcher at ISP RAS, Professor at MSU. Research interests: static analysis, program optimization, parallel programming.