Подход к обнаружению ошибок несоответствия типов в коде на динамических языках программирования

Бронштейн И. Е.
ibronstein@ispras.ru

Аннотация. Статья посвящена обнаружению дефектов в коде, написанном на динамических языках программирования. Производится обзор статических анализаторов программ на языках Python, Ruby и JavaScript. Показывается, что большинство существующих средств не в состоянии обнаруживать целый класс дефектов: ошибки несоответствия типов. Даётся определение таких ошибок, приводятся данные о том, что они весьма распространены, а также критичны по мнению разработчиков программ. Предлагается подход к выводу типов для динамических языков, а также к реализации обнаружителей дефектов на его основе. Вводится понятие трассы дефекта, описывается построение такой трассы.

Ключевые слова: статический анализ; обнаружение дефектов; динамическая типизация данных; Python; Ruby; JavaScript; несоответствие типов; трассы дефектов.

1 Введение

Среди существующих языков программирования можно выделить большую группу *динамических языков*. Главный фактор, определяющий, является ли язык динамическим, — используемая в языке типизация данных [1]. Статическая типизация данных означает, что контроль типов осуществляется во время компиляции. Напротив, при динамической типизации проверки производятся в процессе выполнения программы. В этом случае тип имеется у тех или иных значений, однако у переменных он как таковой отсутствует. В процессе работы программы одной и той же переменной могут быть присвоены значения различных типов [2].

Несмотря на отсутствие единого определения динамических языков программирования, обычно к ним относят языки, которые поддерживают: (1) динамическую типизацию данных; (2) изменение кода программы во время выполнения (например, добавление полей класса, изменение иерархии типов и

т. д.), а также выполнение произвольного кода «на лету» (функция `eval`). Как правило, динамические языки являются интерпретируемыми.

В последнее время такие языки получили большое распространение. Согласно индексу TIOBE по состоянию на август 2013 года [3], в десятку самых популярных языков программирования входят PHP, Python, JavaScript и Ruby, являющиеся динамическими языками. Perl также не сильно уступает им в популярности. Всё чаще используются фреймворки, включающие в себя динамические языки: AJAX (включает JavaScript), LAMP (Perl, PHP или Python), Ruby on Rails (Ruby). В последние годы динамические языки стали поддерживаться основными производителями интегрированных средств разработки (IDE). Популярность динамических языков можно проследить на примере языка Python: в последнее время он стал использоваться не только для написания небольших скриптов, но и для создания крупных программных продуктов. Исходный код таких проектов, как TACTIC, Twisted, Django и SQLAlchemy, содержит более сотни тысяч строк, что весьма велико, учитывая, насколько высокоуровневыми являются программные конструкции языка Python.

2 Обзор статических анализаторов для динамических языков

В связи со всё возрастающей сложностью программ, разрабатываемых на динамических языках, остро стоит проблема их надёжности. Чтобы решить эту проблему для программ на статических языках (например, Си/Си++ и Java), активно используются средства, позволяющие проводить анализ программ с целью обнаружения тех или иных дефектов. Для выявления ошибок реализации чаще всего используются методы статического анализа, осуществляемого по исходному коду анализируемой программы без её запуска. Исследования на конкретных программных продуктах показывают, что сложность исправления ошибки, обнаруженной при помощи статического анализа, ниже, чем если бы эта ошибка была найдена на более поздних этапах разработки [4].

Так, для программ на языках Си и Си++ существует множество профессиональных статических анализаторов (Svace [5], Coverity Prevent, Klocwork Insight, PVS-Studio и другие). Эти средства включают в себя набор автоматических обнаружителей дефектов (чекеров), которые могут: (1) выполняться после стадии семантического анализа путём обхода синтаксического дерева (AST) и анализа навешанных на это дерево атрибутов (без анализа путей выполнения); (2) осуществлять анализ потоков данных (dataflow) по промежуточному представлению программы. Эти две основные группы можно назвать простыми и, соответственно, сложными чекерами. С помощью сложных чекеров возможно обнаруживать критические дефекты: выход за границы массива, разыменование нулевого указателя, использование освобождённой памяти или значения неинициализированной переменной.

Для программ на динамических языках программирования также, хотя и в меньшем количестве, существуют статические анализаторы. Подавляющее большинство среди них составляют программные средства, не использующие вывод типов: например, Pyflakes [6] и Pylint [7] (для языка Python), reek [8] и ruby-lint [9] (для языка Ruby), JSLint [10] и JSHint [11] (для языка JavaScript). Все эти инструменты объединяет то, что в них реализован набор чекеров, выполняющихся путём обхода синтаксического дерева без навешанных на него атрибутов. В качестве примера можно привести нахождение операторов без побочного эффекта. Ниже приведены код на языке Python и предупреждение, которое выводится при анализе данного кода анализатором Pylint:

```
x = 0

x
```

```
$ pylint true_positive1.py

W:  2, 0: Statement seems to have no effect

(pointless-statement)
```

В целом, реализованные в вышеперечисленных средствах чекеры направлены на поиск в анализируемом коде некоторых анти-паттернов программирования. Иногда также проверяется, соответствует ли программа принятым в языке стандартам кодирования (например, PEP8 [12] для языка Python). Проводя аналогию с Си++, описанные чекеры можно назвать простыми. Они чаще всего соответствуют не сообщениям об ошибках, а предупреждениям, который выдавал бы компилятор статического языка программирования.

Хотя про выше названные инструменты сказано, что в них не реализован вывод типов, это не совсем так. В Pylint частично реализован вывод типов, на основе которого осуществляется проверка, можно ли использовать объект в выражении «вызов функции». Однако, вывод типов в Pylint является локальным (производится лишь в рамках одной функции) и неполным (поддерживает не все возможные выражения). Так, для следующего примера инструмент выдаст сообщение об ошибке:

```
x = 0

x()
```

```
$ pylint true_positive2.py
E:  2, 0: x is not callable (not-callable)
```

Однако, если пример немного усложнить (назовём получившийся пример `false_negative.py`), ошибка не будет обнаружена, поскольку вывод типов в Pylint не сопоставляет параметры функции и аргументы её вызова:

```
def foo(p): p()
x = 0
foo(x)
```

```
$ pylint false_negative.py
$
```

3 Ошибки несоответствия типов

Как видно, код в `false_negative.py` содержит ошибку, которая не обнаруживается ни одним из средств, представленных выше. Обобщая ситуацию, можно заметить, что поскольку в этих средствах не реализован вывод типов, они не могут обнаруживать *ошибки несоответствия типов*. Говоря неформально, под такими ошибками мы будем понимать непредусмотренный разработчиком вызов функции или операции (например, доступа к атрибуту объекта) с аргументами (операндами) типов, которые функцией (операцией) не поддерживаются. Ошибка в `false_negative.py` подходит под это определение, поскольку выражение `x()` можно считать неявным обращением к `x.__call__` (аналог метода `operator()` в Си++). К ошибкам несоответствия типов можно отнести и ситуацию, когда число параметров функции и число аргументов её вызова не совпадают.

Возникает вопрос, насколько важными являются именно ошибки несоответствия типов, и, следовательно, насколько оправдано решать задачу вывода типов, необходимого для их обнаружения. Чтобы ответить на этот вопрос, сначала определим исследуемый класс ошибок более строго.

Номер	Тип дефектов	Описание	Исправление
010	Документация	Комментарии, сообщения	Изменение документации, комментариев и т. д.
020	Синтаксис	Синтаксические ошибки, опечатки в коде и т. д.	Исправление синтаксической или иной ошибки во время компиляции
030	Сборка, пакеты	Подключение библиотек, контроль версий и т. д.	Создание или использование правильной версии
040	Присваивания	Декларации, дубликаты имён, области видимости имён и т. д.	Исправление оператора
050	**Интерфейс**	**Вызовы процедур, ввод/вывод, пользовательские форматы**	**Изменение интерфейса**
060	Проверки	Сообщения об ошибках, некорректные проверки	Добавление или исправление обработки ошибок
070	Данные	Структура, содержимое	Изменение программы для поддержания целостности данных
080	Функция	Логика, указатели, циклы, рекурсия, вычисления и т. д.	Изменение совокупности операторов
090	Система	Конфигурация, хронометраж, память	Адаптация программы к внешним факторам (например, ОС)
100	Окружение	Ошибки в системе поддержки: компиляторе, тестовых данных и т. д.	Исправление ошибки компилятора или отказ от его использования, исправление тестовых данных

Таблица 1. Общая DTS-классификация.

Рассмотрим стандарт Personal Software Process Defect Type Standard (PSP DTS) — распространённую классификацию программных дефектов [13,14]. В ней описаны десять различных видов ошибок, которые обычно встречаются в программных продуктах. Для каждого вида перечислены его номер (от 010 до 100), название (тип дефектов), описание, а также исправление, которое необходимо внести для устранения ошибок данного вида. Общая (независимая от языка программирования) DTS-классификация представлена в таблице 1.

Номер	Тип дефектов	Описание
010	Документация	Ошибки в документационных строках и комментариях
020	Синтаксис	SyntaxError, IndentationError
030	Стандарт кодирования	Предупреждения и ошибки, связанные с PEP8
040	Присваивание	NameError (нет объявления), IndexError/KeyError,
		UnboundLocalError (область видимости), ImportError
050	**Интерфейс**	**TypeError, AttributeError: неправильные параметры и методы**
060	Проверки	AssertionError (ошибка в assert), ошибки в тестах (doctest или unittest)
070	Данные	ValueError (неправильные данные), ArithmeticError,
		EOFError, BufferError
080	Функция	RuntimeError и логические ошибки
090	Система	SystemError (MemoryError, ReferenceError)
100	Окружение	EnvironmentError: ошибки ОС и сторонних библиотек, ошибки ввода/вывода (IOError)

Таблица 2. DTS-классификация для языка Python.

Как следует из общей DTS-классификации и определения ошибок несоответствия типов, последние соответствуют DTS-ошибкам вида 050 («интерфейс»). В терминах DTS несоответствие типов — это неправильное использование интерфейса, а исправление такой ошибки — это изменение либо интерфейсной функции, либо способа её использования (передаваемых в функцию аргументов).

Также существуют модификации DTS применительно к различным языкам программирования. В этой связи интерес представляет DTS-классификация для языка Python [15], представленная в таблице 2. В качестве описания для большинства видов дефектов приведены типы (классы) исключений, соответствующие данному виду дефектов. Ошибкам вида «интерфейс»

соответствуют исключения `AttributeError` и `TypeError`. Таким образом, с формальной точки зрения, ошибка несоответствия типов в языке Python — это ситуация, когда в результате вызова некоторой функции или метода возбуждается исключение типа `AttributeError` или `TypeError`, причём обработки этого исключения выше по стеку не происходит (это и означает, что вызов не был предусмотрен разработчиком), что приводит к аварийному завершению программы.

Для других динамических языков можно дать аналогичные определения с той лишь разницей, что типы исключений здесь будут другими. Так, для Ruby речь будет идти об исключениях классов `NoMethodError` и `TypeError`, для JavaScript — класса `TypeError`.

Наличие формального определения для ошибок несоответствия типов позволяет определить, насколько они важны, с помощью количественных критериев. Будем рассматривать два следующих критерия:

- долю отчётов об ошибках несоответствия типов среди всех отчётов об ошибках (распространённость);
- долю критичных отчётов среди отчётов об ошибках несоответствия типов по сравнению с долей критичных отчётов целом (критичность).

В качестве источника отчётов выберем системы отслеживания ошибок известных проектов с открытым исходным кодом на языке Python: Exaile [16], calibre [17], SQLAlchemy [18], Bazaar [19], Twisted [20], Trac [21] и Django [22].

Название проекта	**Все отчёты об ошибках**	**AttributeError и TypeError**	**Доля, %**
Exaile	1350	86	**6,37**
calibre	1357	77	**5,67**
SQLAlchemy	2826	230	**8,14**
Bazaar	4646	167	**3,46**
Twisted	6664	197	**2,96**
Trac	10472	627	**5,99**
Django	20664	969	**4,69**

Таблица 3. Распространённость ошибок несоответствия типов.

Данные по распространённости ошибок несоответствия типов приведены в таблице 3. Из таблицы видно, что доля отчётов, посвящённых именно ошибкам несоответствия типов, для некоторых проектов превышает 8 %. При этом необходимо учитывать, что тема значительной части отчётов (более 90 %) —

не дефекты, а логические ошибки, которые обнаружить автоматическими средствами невозможно [2]. Таким образом, отчёты об ошибках несоответствия типов составляют для некоторых проектов большинство отчётов об ошибках, которые можно найти автоматически.

Название проекта	Критичность всех отчётов, %	Критичность AE/TE, %	Более критичны
Exaile	8,88	11,63	Д
Bazaar	4,72	4,25	Н
Trac	5,36	10,03	Д
Django	1,95	2,68	Д

Таблица 4. Критичность ошибок несоответствия типов.

Данные по критичности ошибок несоответствия типов приведены в таблице 4. Критичными считались отчёты с большим значением поля «критичность», если таковое присутствовало в системе отслеживания ошибок: «Critical» и «High» в поле «Importance» (для Exaile), «Critical» в поле «Importance» (для Bazaar), «blocker» и «critical» в поле «Severity» (для Trac), «Release blocker» в поле «Severity» (для Django). Для всех проектов, кроме Bazaar, критичность ошибок несоответствия типов выше, чем в среднем. Заметим, однако, что 15 % кода Bazaar написано на языке Си. Это могло послужить причиной более низкой критичности ошибок несоответствия типов для этого проекта.

4 Обнаружение ошибок в программах на динамических языках

Таким образом, ошибки несоответствия типов широко распространены и, по мнению самих разработчиков проектов, достаточно критичны. Для обнаружения таких ошибок необходимо наличие глобального и полного (в смысле поддержки программных конструкций того или иного динамического языка) вывода типов. Отметим, что, несмотря на отсутствие такого вывода типов в перечисленных выше статических анализаторах, исследовательские проекты по реализации вывода типов для различных динамических языков всё же существуют. К подобным проектам можно отнести DRuby [23] (для языка Ruby) и TAJS [24] (для языка JavaScript).

В данной работе представляется подход к построению статического анализатора, обнаруживающего ошибки несоответствия типов на основе информации, полученной на этапе вывода типов. Для вывода типов предлагается использовать алгоритмы на графе типовых переменных. Работа с таким графом во многом аналогична другим методам вывода типов, например, используемым в проекте DRuby уравнениям, задающим ограничения на типы

выражений [25]. Однако, на наш взгляд, представление ограничений именно в виде графа значительно упрощает построение трасс дефектов, о которых будет сказано несколько позже.

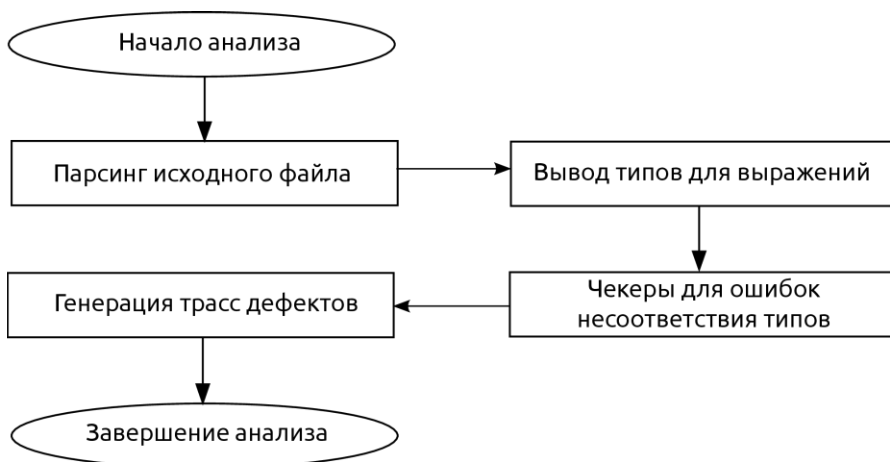

Рис. 1. Общая схема работы статического анализатора.

Общая схема работы статического анализатора в рамках предлагаемого подхода представлена на рис. 1. Ниже мы более подробно разберём все этапы работы за исключением парсинга исходного кода. Задача парсинга, как правило, не представляет особой сложности и может быть решена как стандартными средствами языка (например, модуль `ast` для языка Python), так и при помощи сторонних библиотек (например, jruby-parser [26] для языка Ruby или Esprima [27] для языка JavaScript).

5 Вывод типов для выражений

В предлагаемом подходе к построению статического анализатора для вывода типов используется представление программы в виде графа типовых переменных. Каждому выражению в программе ставится в соответствие свой узел графа: типовая переменная, то есть переменная, значением которой является множество типов. Изначально типовые переменные хранят в себе пустые множества типов, за исключением констант и строковых литералов, которые в качестве значения получают множество из одного единственного типа: типа константы (литерала). Множество возможных типов для выражения `expr` обозначается `type(expr)`.

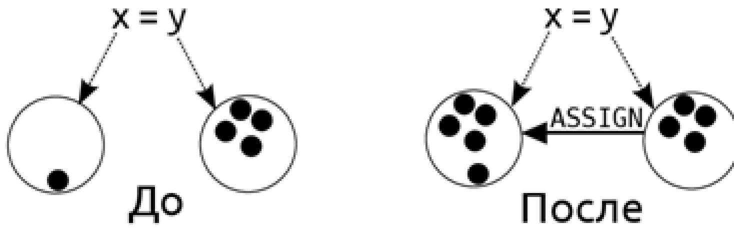

Рис. 2. Ситуация до и после добавления дуги и пропагации по ней типов.

Граф типовых переменных является ориентированным. Дуги графа, называемые также ограничениями на типы, создаются при обходе тех или иных языковых конструкций в программном коде. Дуги помечаются названием: видом ограничения, который зависит от того, какая конструкция анализируется (например, для присваивания соответствующее ограничение будет иметь вид `ASSIGN`). Типы распространяются (пропагируются) по графу типовых переменных по определённым правилам, которые устанавливаются ограничениями. Так, если в программе выполняется присваивание `x = y`, то любой тип, который может принимать выражение `y`, может также иметь и переменная `x`. Поэтому ограничение вида `ASSIGN` пропагирует все типы из `y` в `x`, отражая тот факт, что $\mathtt{type(y)} \subseteq \mathtt{type(x)}$. Это иллюстрирует рис. 2 (типы изображены на нём в виде чёрных точек).

Добавление в граф новых ограничений создаёт необходимость в дальнейшей пропагации типов. Верно и обратное: например, для вызова некоторого метода в результате пропагации может быть вычислен новый тип объекта. Значит, возможно, будет необходимо анализировать тела методов в этом объекте, а при анализе тел методов могут быть созданы новые дуги в графе (ограничения) и так далее. В алгоритмах, которые работают с графом типовых переменных, циклически создаются новые ограничения и пропагируются типы до тех пор, пока эти действия вносят изменения в граф. Работа с графом типовых переменных зависят от семантики того или иного динамического языка программирования. Виды ограничений, создаваемых для программ на языке Python, подробно описаны в статье [2], которая служит основой для исследовательского проекта TIRPAN [28].

В выше упомянутой работе также предлагается подход к анализу функций, написанных не на языке Python и таким образом являющихся внешними по отношению к анализируемой программе. Суть подхода, который можно распространить и на другие динамические языки программирования, в следующем: поведение внешних функций моделируется в статическом анализаторе при помощи *спецификаций*. Спецификации — это специальные функции, которые принимают на вход типы параметров исходной функции, а

на выходе по ним строится тот тип, который имело бы возвращаемое значение исходной функции. Также, возможно, моделируется внесение изменений в типы входных параметров (например, для функции `append` в языке Python или аналогичной ей функции `push` в языке Ruby тип первого параметра меняется в ходе выполнения).

6 Чекеры для ошибок несоответствия типов

Стадия работы чекеров следует непосредственно после вывода типов. Входными данными для чекеров является AST с навешанными на него типовыми переменными, хранящими информацию о типах выражений. Однако, в данной работе предполагается, что чекеры работают с AST не напрямую, а посредством специально предназначенной для этого прослойки: фреймворка для чекеров. Ниже мы опишем функциональность этого фреймворка.

Фреймворк осуществляет итерирование по AST, при этом пропускаются участки кода, для которых выше по стеку есть обработчики исключений, соответствующих ошибкам несоответствия типов для данного языка. Например, для следующего кода на языке Ruby фреймворк пропустит код с вызовом `getTranslation`, иначе это могло бы привести к многочисленным ложным срабатываниям чекера, проверяющего доступ к несуществующему методу:

```
begin
    translation = message.getTranslation()
rescue NoMethodError
    translation = message
end
```

Фреймворк предоставляет чекерам возможность зарегистрировать произвольное число функций обратного вызова (callback), которые будет вызываться при входе в узлы или выходе из узлов определённого типа. Предположим, что для кода на языке Python необходимо реализовать чекер `basenameChecker`, проверяющий, что в функцию `os.path.basename` было передано значение, отличное от строки. Регистрация такого чекера может выглядеть следующим образом:

```
def basenameChecker():
  registerCallback(Event.ON_ENTER, ast.Call,
processBasenameCall)
```

Основная функциональность фреймворка — это предоставление интерфейсных функций следующих видов: (1) функций, возвращающих множество типов для узла (например, `getNodeTypes`); (2) предикатов для проверки типа (например, `isFunction`); (3) функций для работы с внутренней структурой типа или для вывода его внешнего представления (например, `getListElements` или `getQualifiedName`); (4) функций формирования сообщения об ошибке (например, `reportError`). Такой интерфейс позволяет реализовать каждую из функций, из которых состоит чекер несоответствия типов, в компактном виде. Например, реализация `processBasenameCall` с использованием интерфейсных функций может выглядеть так:

```
def isBasenameFunction(element):
    return (isFunction(element) and
            getQualifiedName(element) ==
"os.path.basename")


def processBasenameCall(node):
    functionTypes = node.func.getNodeTypes()
    if any(isBasenameFunction(type) for type in
functionTypes):
        argTypes = node.args[0].getNodeTypes()
        for type in argTypes:
            if not isString(type):
                reportError(Defect.BASENAME,
node, type)
    return TRAVERSE_ALL
```

7 Генерация трасс дефектов

Предлагаемый подход к построению статического анализатора предполагает генерацию трасс для найденных ошибок несоответствия типов. Говоря неформально, трасса дефекта — это описание последовательности шагов, из которой можно понять, как неправильный тип попадает в точку несоответствия типов. Такое описание должно помочь пользователю статического анализатора определить, является ли обнаруженный дефект истинным. Чтобы понять важность наличия трассы, рассмотрим ошибку несоответствия типов из проекта Gramps [2, 29], которая заключается в том, что в функцию `os.path.basename` вместо строки передаётся пустой список:

```
class HtmlDoc:
...
    def set_css_filename(self, css_filename):
        if os.path.basename(css_filename):
            self.css_filename = css_filename
        else:
            self.css_filename = ''


class DocReportDialog:
    def __init__(self):
        self.doc = HtmlDoc(...)
        self.CSS =
PLUGMAN.process_plugin_data("WEBSTUFF")
        self.css_filename =
self.CSS[id]["filename"]
...
    def make_document(self):

self.doc.set_css_filename(self.css_filename)
```

```
...
def on_ok_clicked(...):
    dialog = DocReportDialog(...)
    dialog.make_document()
```

При обнаружении в `set_css_filename` дефекта несоответствия типов чекер `basenameChecker` может предоставить пользователю трассировку стека (stack trace), включающую в себя функции `set_css_filename`, `make_document`, `on_ok_clicked` и так далее. Однако, такая трассировка малоинформативна, поскольку не позволяет проследить, откуда у `css_filename` появилось значение, не являющееся строкой. Эту информацию можно получить путём «обратного разбора» функции `process_plugin_data`, что, с свою очередь, потребует дальнейшего анализа. Подобный анализ и является генерацией трассы дефекта.

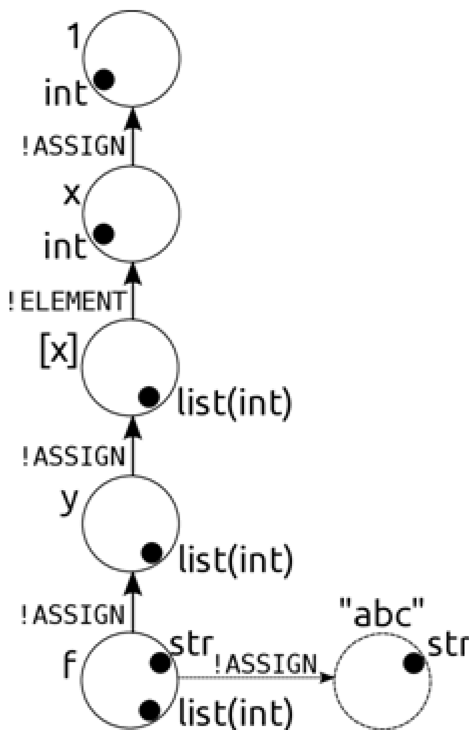

Рис. 3. «Обратный» граф типовых переменных.

Так как типовые переменные являются вершинами ориентированного графа, понятие трассы дефекта можно определить более формально, рассмотрев новый граф, в котором все вершины соединены обратными дугами. Это означает, что, если в исходном графе имелось ограничение вида ASSIGN от переменной A к переменной B, в новом графе будет присутствовать дуга от B к A. Будем считать, что обратные дуги имеют вид «конкатенация ! и названия вида первоначальной дуги», например, !ASSIGN.

После того, как мы ввели новый граф, трассу можно определить как набор путей в нём, причём каждый путь начинается в переменной, соответствующей месту дефекта (например, аргументу os.path.basename для Defect.BASENAME) и строится по определённым правилам, которые задаются дугами в графе. Рассмотрим небольшой участок кода и покажем на примере, как при помощи его «обратного» графа (показан на рис. 3) для дефекта Defect.BASENAME строится трасса, состоящая из ровно одного пути:

```
x = 1

y = [x]

if condition(): f = y

else: f = "abc"

print os.path.basename(f)
```

Построение пути начинается с типовой переменной для f, которая становится текущей вершиной для алгоритма. Для каждого узла в алгоритме задаётся текущее множество типов, поиск которых осуществляется. Первоначально (для первой вершины пути) это множество равно множеству «неправильных» типов для дефекта. Для данного примера множество равно {list(int)}. Далее производится добавление в трассу новых вершин графа в соответствии с ограничениями и обход вершин в глубину. Ограничение вида !ASSIGN добавляет в граф вершину на конце дуги (назовём её $V_{!ASSIGN}$) в случае, если пересечение множества искомых типов для текущего узла и $type(V_{!ASSIGN})$ не пусто. Когда $V_{!ASSIGN}$ станет текущей вершиной, множеством искомых типов для неё будет это пересечение. Согласно этому правилу, переменная для "abc" в трассу не добавляется (поэтому она и дуга к ней изображены на рис. 3 пунктиром), а переменные для y и [x] добавляются. Правило для вершин на конце !ELEMENT также вычисляет пересечение типов, но при этом «раскрываются» коллекции (например, {list(int)} превращается в {int}). По этому правилу в трассу добавляется переменная для x и, следовательно, для 1. На этом построение трассы заканчивается, поскольку вершин для добавления в неё больше нет.

8 Заключение

В работе предложен подход к построению статического анализатора, обнаруживающего в коде на динамических языках программирования ошибки несоответствия типов и генерирующего трассы для найденных дефектов. При этом описанные методы не привязаны к специфике какого-либо конкретного динамического языка.

Перечислим возможные направления дальнейших исследований по рассматриваемой тематике:

1. Исследование того, насколько чувствительность к потоку выполнения важна для повышения точности анализа (в настоящий момент алгоритмы, работающие с графом типовых переменных, нечувствительны к потоку выполнения).
2. Добавление поддержки инкрементального анализа, при котором небольшие изменения в тексте программы вызывают пересчёт небольшого числа типов выражений, что позволило бы выполнять статический анализ при наборе кода в IDE.
3. Организация обратной связи, которая позволила бы использовать при анализе не только вычисленную информацию, но и некоторые подсказки от пользователя о возможных типах выражений в программе.

Список литературы

[1]. L. D. Paulson. Developers shift to dynamic programming languages. IEEE Computer, vol. 40, issue 2, February 2007, pp. 12–15.

[2]. Бронштейн И. Е. Вывод типов для языка Python. Труды Института системного программирования РАН, том 24, 2013, стр. 161–190.

[3]. TIOBE Programming Community Index for August 2013: http://tinyurl.com/tiobe-201308

[4]. Савицкий В. О., Сидоров Д. В. Инкрементальный анализ исходного кода на языках C/C++. Труды Института системного программирования РАН, том 22, 2012, стр. 119—129.

[5]. Аветисян Арутюн, Белеванцев Андрей, Бородин Алексей, Несов Владимир. Использование статического анализа для поиска уязвимостей и критических ошибок в исходном коде программ. Труды Института системного программирования РАН, том 21, 2011, стр. 23–38.

[6]. Pyflakes: https://launchpad.net/pyflakes

[7]. Pylint - code analysis for Python: http://www.pylint.org

[8]. reek: https://github.com/troessner/reek

[9]. ruby-lint: http://code.yorickpeterse.com/ruby-lint/latest

[10]. JSLint: http://www.jslint.com

[11]. JSHint, a JavaScript Code Quality Tool: http://www.jshint.com

[12]. PEP 8 – Style Guide for Python Code: http://www.python.org/dev/peps/pep-0008

[13]. R. Chillarege, I. Bhandari, J. Chaar, M. Halliday, D. Moebus, B. Ray, M. Y. Wong. Orthogonal Defect Classification — A Concept of In-Process Measurements (1992).

[14]. W. Humprey. A discipline for software engineering (1997).

[15]. M. Reingart. Rad2py: Plataforma de Trabajo para el Desarrollo Rápido de Aplicaciones bajo un Proceso de Software Personal, pp. 152–153: http://code.google.com/p/rad2py/wiki/DefectTypeStandard
[16]. Bugs : Exaile: https://bugs.launchpad.net/exaile
[17]. Bugs : calibre: https://bugs.launchpad.net/calibre
[18]. sqlalchemy: http://www.sqlalchemy.org/trac
[19]. Bugs : Bazaar: https://bugs.launchpad.net/bzr
[20]. Twisted: http://twistedmatrix.com/trac
[21]. The Trac Project: http://trac.edgewall.org
[22]. Django: https://code.djangoproject.com
[23]. DRuby - Home: http://www.cs.umd.edu/projects/PL/druby
[24]. Type Analysis for JavaScript: http://www.brics.dk/TAJS
[25]. Michael Furr, Jong-hoon (David) An, Jeffrey S. Foster, Michael Hicks. Static type inference for Ruby. SAC '09 Proceedings of the 2009 ACM symposium on Applied Computing, pp. 1859–1866.
[26]. jruby-parser: https://github.com/jruby/jruby-parser
[27]. esprima: https://github.com/ariya/esprima
[28]. tirpan: https://github.com/bronikkk/tirpan
[29]. Gramps Bug Report 005023: http://www.gramps-project.org/bugs/view.php?id=5023

Approach to detecting types inconsistency errors in a program code in dynamic languages

I. E. Bronshteyn
ibronstein@ispras.ru

Abstract. The paper deals with detection of defects in a program code written in dynamic languages. At first, overview of static analyzers for programs in Python, Ruby and JavaScript is done. After this, the paper shows that most of existing tools are not able to detect entire class of defects: types inconsistency errors. Such errors are defined, the paper proves that the errors are prevailing and rather critical in the opinion of software developers. It presents an approach to type inference for dynamic languages and to implementation of checkers based on output from type inference. Concept of defect trace is introduced and construction of such trace is described then.

Keywords: static analysis; defects detection; dynamic typing; Python; Ruby; JavaScript; types inconsistency; defects traces.

References

[1]. L. D. Paulson. Developers shift to dynamic programming languages. IEEE Comp vol. 40, issue 2, February 2007, pp. 12–15.
[2]. I. E. Bronstein Type inferrence for Python prigramming language. rudy ISP RAN [The Proceedings of ISP RAS], 2013, vol 24, pp. 161-190 (in Russian)
[3]. TIOBE Programming Community Index for August 2013: http://tinyurl.com/tiobe-201308
[4]. V. O. Savitsky, D. V. Sidorov, Inkremental'nyj analiz iskhodnogo koda na yazykakh C/C++[Incremental source code analysis for C/C++ languages] Trudy ISP RAN [The Proceedings of ISP RAS], 2012, vol 22, pp. 119-129 (in Russian)
[5]. A. Avetisyan, A. Belevantsev, A. Borodin, V. Nesov Ispol'zovanie staticheskogo analiza dlya poiska uyazvimostej i kriticheskikh oshibok v iskhodnom kode programm[Using static analysis for finding security vulnerabilities and critical errors in source code] Trudy ISP RAN [The Proceedings of ISP RAS], 2012, vol 21, pp. 23-38 (in Russian)
[6]. Pyflakes: https://launchpad.net/pyflakes
[7]. Pylint - code analysis for Python: http://www.pylint.org
[8]. reek: https://github.com/troessner/reek
[9]. ruby-lint: http://code.yorickpeterse.com/ruby-lint/latest
[10]. JSLint: http://www.jslint.com
[11]. JSHint, a JavaScript Code Quality Tool: http://www.jshint.com
[12]. PEP 8 – Style Guide for Python Code: http://www.python.org/dev/peps/pep-000
[13]. R. Chillarege, I. Bhandari, J. Chaar, M. Halliday, D. Moebus, B. Ray, M. Y. Wo Orthogonal Defect Classification — A Concept of In-Process Measurements (19 [14] W. Humprey. A discipline for software engineering (1997).
[14]. W. Humprey. A discipline for software engineering (1997).
[15]. M. Reingart. Rad2py: Plataforma de Trabajo para el Desarrollo Rápido de Aplicaciones bajo un Proceso de Software Personal, pp. 152–153: http://code.google.com/p/rad2py/wiki/DefectTypeStandard

[16]. Bugs : Exaile: https://bugs.launchpad.net/exaile
[17]. Bugs : calibre: https://bugs.launchpad.net/calibre
[18]. sqlalchemy: http://www.sqlalchemy.org/trac
[19]. Bugs : Bazaar: https://bugs.launchpad.net/bzr
[20]. Twisted: http://twistedmatrix.com/trac
[21]. The Trac Project: http://trac.edgewall.org
[22]. Django: https://code.djangoproject.com
[23]. DRuby - Home: http://www.cs.umd.edu/projects/PL/druby
[24]. Type Analysis for JavaScript: http://www.brics.dk/TAJS
[25]. Michael Furr, Jong-hoon (David) An, Jeffrey S. Foster, Michael Hicks. Static type inference for Ruby. SAC '09 Proceedings of the 2009 ACM symposium on Applied Computing, pp. 1859–1866.
[26]. jruby-parser: https://github.com/jruby/jruby-parser
[27]. esprima: https://github.com/ariya/esprima
[28]. tirpan: https://github.com/bronikkk/tirpan
[29]. Gramps Bug Report 005023: http://www.gramps-project.org/bugs/view.php?id=5023