Вывод типов для языка Python

Бронштейн И. Е.
ibronstein@ispras.ru

Аннотация. Тема статьи — вывод типов для программного кода на языке Python. Сначала производится обзор описанных в научной литературе алгоритмов вывода типов для языков с параметрическим полиморфизмом. Затем даётся описание нового алгоритма, являющегося модификацией одного из предыдущих: алгоритма декартова произведения. Показывается, как модуль вывода типов, использующий новый алгоритм, анализирует различные конструкции языка Python. Представляются результаты работы над прототипом.

Ключевые слова: python; вывод типов; динамическая типизация данных; статический анализ; обнаружение дефектов.

1. Введение

В современном индустриальном программировании остро стоит проблема надёжности разрабатываемого программного обеспечения. Поэтому существует множество программных средств для обнаружения тех или иных дефектов в анализируемом коде. Анализ, осуществляемый по исходному коду без запуска соответствующей программы, называется *статическим анализом* программного обеспечения. Исследования на конкретных программных продуктах показывают, что сложность исправления ошибки, найденной при помощи статического анализа, ниже, чем если бы эта ошибка была найдена на более поздних этапах разработки [1].

Статические анализаторы существуют в основном для статически типизированных языков программирования (например, Си, Си++ и Java). Однако в настоящее время всё большую популярность завоёвывают языки с динамической типизацией данных. Согласно авторитетному рейтингу от компании TIOBE Software [2], в десятку самых популярных языков программирования входят Objective-C (смешанная типизация) и динамически типизированные PHP, Python, Perl и Ruby (JavaScript — 11-й). Благодаря тому, что динамически

типизированные языки гибки и способствуют быстрой разработке, их всё чаще предпочитают для создания даже весьма крупных программных систем. В то же время понятно, что отсутствие у компилятора информации о типах выражений может приводить к ошибкам времени исполнения, которые для языков со статической типизацией могут быть обнаружены уже на этапе компиляции. Так, для следующего примера кода на динамически типизированном языке Python отсутствие в классе A метода not_present_method будет обнаружено лишь при попытке вызвать метод с таким именем:

```
class A:
    pass
a = A()
a.not_present_method()
```

Чтобы понять, какие виды дефектов — ошибок и уязвимостей — распространены в программном коде на языке Python, мы проанализировали более 150 отчётов, хранящихся в системах отслеживания ошибок (bug trackers) проектов с открытым исходным кодом. В качестве таких проектов были выбраны крупные системы с сотнями тысяч строк кода: Django, Gramps и Twisted.

Подавляющее большинство отчётов об ошибках в таких системах пришлось исключить из рассмотрения, поскольку речь в них идёт о логических ошибках, а не о дефектах, которые возможно найти при помощи статического анализа. Общими логическими ошибками являются, например, несоответствия между документацией определённого программного модуля и его реальной функциональностью. Примерами частных логических ошибок могут служить неправильные регулярные выражения для проверки пользовательских данных или нуждающийся в корректировке текстовый вывод программы.

Всё остальное, т. е. найденные на реальных проектах дефекты, можно разделить на две основные категории:

- «простые» дефекты — те дефекты, которые можно обнаружить без вывода типов;
- «сложные» дефекты — те дефекты, найти которые можно, лишь используя информацию, полученную при выводе типов.

В табл. 1 приведены данные об общем количестве проанализированных отчётов об ошибках и числе найденных, соответственно, «простых» и «сложных» дефектов на выбранных проектах.

Табл. 1. Данные о найденных дефектах

Название проекта	**Все отчёты об ошибках**	**«Простые» дефекты**	**«Сложные» дефекты**
Django	**135**	3	7
Gramps	**3**	0	1
Twisted	**16**	0	2

Как видно из таблицы, для языка Python большинство ошибок, которые теоретически могут быть обнаружены с помощью статического анализа, составляют именно ошибки несоответствия типов. Статический анализатор сможет находить такие ошибки, только если в нём имеется реализованный модуль вывода типов. Однако статических анализаторов для динамически типизированных языков, где присутствовал бы нетривиальный вывод типов, нам найти не удалось. Между тем, анализ одного из отчётов об ошибке, найденного в системе отслеживания ошибок проекта Gramps [3], показал: модуль вывода типов должен поддерживать большинство основных конструкций языка Python, чтобы на основе такого вывода типов можно было обнаружить дефект, о котором идёт речь в отчёте: дефект несоответствия типов. Должны поддерживаться как минимум следующие элементы языка Python: стандартные структуры данных; основные управляющие конструкции; импорт модулей; классы/объекты и их методы; некоторые встроенные функции для стандартных типов; некоторые бинарные операции; атрибуты объектов; функции, используемые как параметры других функций; исполнение программного кода из файлов; анонимные (лямбда-) функции.

Целью данной работы является реализация вывода типов для программного кода на языке Python. Под выводом типов подразумевается модуль, в результате работы которого каждому выражению в программном коде на языке Python будет поставлено в соответствие множество возможных типов для этого выражения. При этом, чтобы продемонстрировать работоспособность полученной реализации, планируется использовать вышеупомянутый пример несоответствия типов из проекта Gramps.

2. Обзор существующих решений

В [4] даётся обзор алгоритмов вывода типов для программного кода, в котором присутствует *параметрический полиморфизм*. Под параметрическим полиморфизмом понимается способность функции или метода работать со значениями различных типов (например, функция len в Python может принимать на вход различные коллекции и возвращать их размер). В работе показано развитие алгоритмов вывода

типов (на примере динамически типизированного языка программирования Self) от самого простого алгоритма (т. н. «базового» алгоритма — basic algorithm) до алгоритма декартова произведения (АДП). Мы рассмотрим достаточно подробно только «базовый алгоритм» и АДП.

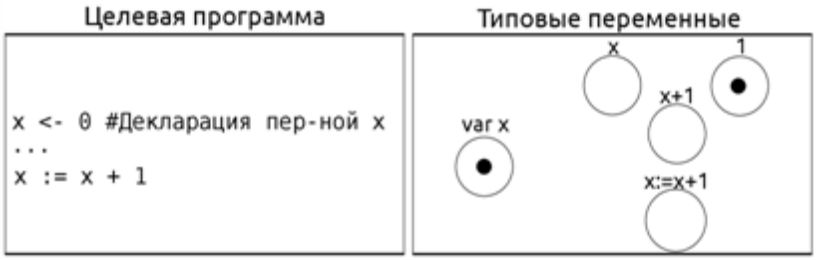

Рис. 1. Фрагмент анализируемой программы на языке Self (слева) и созданные для него типовые переменные (справа). Типовая переменная для декларации x помечена как var x, чтобы отличать её от выражения x (операнда сложения).

«Базовый алгоритм», или алгоритм Палсберга — Шварцбаха был впервые представлен в [5] как решение некоторой системы ограничений, накладываемых на типы выражений в целевой программе. В [4] алгоритм рассматривается в несколько другой, операционной форме: как последовательность шагов, в ходе выполнения которых осуществляется анализ одновременно потока выполнения и потока данных программы. Такой подход позволяет провести прямое соответствие между исполнением программы и её анализом (выводом типов), что упрощает описание алгоритма. Поэтому мы также будем придерживаться операционного подхода. Согласно нему «базовый алгоритм» состоит из трёх шагов:

1. **Создание типовых переменных**. Первый шаг заключается в том, что всем переменным и выражениям в целевой программе ставятся в соответствие *типовые переменные* (см. рис. 1), т. е. переменные, значениями которых являются множества типов. Сначала типовые переменные хранят в себе пустые множества типов, а на следующих двух шагах в них добавляются типы, соответствующие переменным и выражениям, с которыми переменные связаны. Типы из типовых переменных не удаляются.

2. **Инициализация типовых переменных**. На втором шаге в типовые переменные добавляются определённые типы — переменные инициализируются. Цель — зафиксировать первоначальное состояние целевой программы, «засевая» типовые переменные, относящиеся к выражениям или переменным, где некоторые объекты можно найти с самого начала. Так, если в коде программы объявлена переменная, в соответствующую типовую переменную добавляется тип инициализатора. Например, для декларации x <- 0 переменная var x получает значение {*int*}.

Аналогично для строкового литерала ’abc’ в его типовую переменную добавляется тип str. На рис. 2 тип int, добавленный в типовые переменные var x и 1, показан в виде чёрных кружков.

3. **Установка ограничений и пропагация типов**. На последнем шаге строится ориентированный граф, вершинами которого являются типовые переменные. Вершины добавляются поочерёдно и отражают ограничения на типы. Ограничение фактически представляет собой аналог потока данных, только не во время исполнения программы, а на этапе её анализа (вывода типов). Так, если в программе выполняется присваивание x := y, то поток данных направлен из y в x. На интуитивном уровне это означает, что любой тип, который может принимать выражение y, может также иметь и переменная x. Поэтому, когда модуль вывода типов доходит до этого присваивания, он добавляет ребро от типовой переменной для y к типовой переменной для x, отражая тот факт, что type(y) ⊆ type(x). Это иллюстрирует рис. 2.

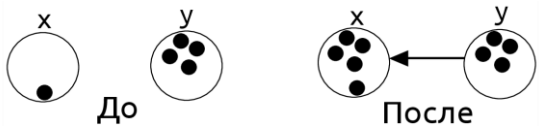

Рис. 2. Ситуация до и после добавления ребра, соответствующего ограничению для x := y, а также пропагация типов.

Всякий раз, когда в граф добавляется ребро, типы пропагируются (распространяются) вдоль него. На рис. 2 показано, как типы из узла y распространяются по ребру и копируются в узел x. По мере того, как в графе появляются новые рёбра, соответствующие ограничениям, типы, которые изначально были лишь в инициализированных типовых переменных, пропагируются всё дальше и дальше. Пропагация устроена таким образом, что ограничения на типы, такие, как type(y) ⊆ type(x), сохраняются после каждого её этапа: если в type(y) добавляются новые типы, они сразу же пропагируются в type(x), и таким образом отношение «является подмножеством» продолжает выполняться.

Добавление новых ограничений создаёт необходимость в дополнительной пропагации. 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 рёбра в графе (ограничения) и т. д. Поэтому на третьем шаге циклически создаются новые ограничения и пропагируются типы до тех пор, пока эти действия вносят какие-либо изменения в граф типовых переменных.

Мы рассмотрели только случай присваиваний, но ограничения на типы должны генерироваться и для других языковых конструкций, например:

- для выражений вида var, где происходит чтение значения некоторой переменной, генерируется ребро от типовой переменной, соответствующей var, к типовой переменной, соответствующей выражению;
- для вызовов функций, благодаря использованию специальных *шаблонов для функций*, генерируются рёбра от аргументов вызова к соответствующим параметрам функции, а от операторов return в вызываемых функциях генерируются рёбра к вызовам.

Последняя фраза нуждается в пояснениях. Шаблоном для функции F называется подграф графа типовых переменных, состоящий из:

- вершин (типовых переменных), соответствующих выражениям, локальным переменным и параметрам F;
- рёбер (ограничений), инцидентных этим вершинам.

Пусть имеется функция max:

```
def max(x, y):
    if x > y:
        return x
    else:
        return y
```

Шаблон для этой функции изображён на рис. 3. На нём также показано, какие рёбра к шаблону создаются при вызове max(3, 4). Для типовой переменной «вызов функции» её значение вычисляется путём пропагации типов аргументов вызова по шаблону для вызываемой функции. Если вызываться может несколько функций, итоговый тип — это объединение типов возвращаемых значений для этих функций. Следовательно, для типовой переменной max(3, 4) будет вычислен тип {*int*}.

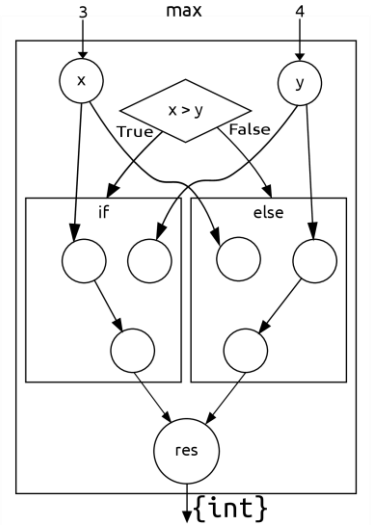

Рис. 3. Шаблон для функции max: сама функция и аргументы её вызова. Тип вызова (тип возвращаемых значений для вызываемых функций) выводится путём пропагации типов аргументов вызова по шаблону.

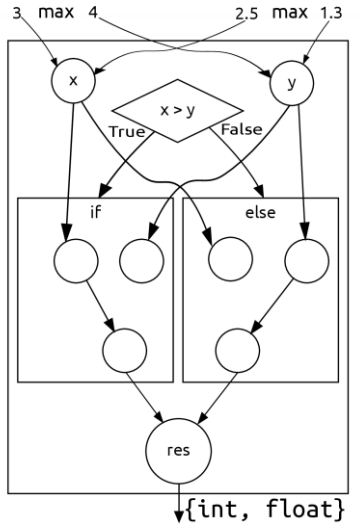

Рис. 4. «Базовый алгоритм» выводит неточный тип {int, float} для двух вызовов max, хотя один из вызовов считает максимум двух целых чисел, а второй — двух вещественных

Проблема «базового алгоритма» в том, как он работает для полиморфных функций, например, для max. На рис. 4 показано, что произойдёт, если в программе присутствует два вызова max: один с целочисленными, а второй с вещественными аргументами. Для обоих вызовов вычисляется тип {*int, float*}. Это неточно, так как очевидно, что вызов max от целых чисел не может вернуть вещественное число и наоборот. Такая неточность вызвана тем, что разные типы — int и float — сливаются на входе в шаблон для функции max. Такое слияние вызывает потерю информации: «склеенные» типы сложно разделить обратно, и в результате получаются неточные типы для всех вызовов одной и той же функции.

Из-за того, что «базовый алгоритм» выводит неточные типы, если в коде присутствует параметрический полиморфизм, Палсберг и Шварцбах представили несколько идей по улучшению алгоритма. Например, было предложено создавать отдельные шаблоны для каждого вызова определённой функции, например, max. Однако, это показало себя крайне неэффективным с точки зрения производительности.

Вместо внесения незначительных улучшений в «базовый алгоритм» в [4] авторы представляют новый алгоритм вывода типов: алгоритм декартова произведения (АДП), который не сливает типы при вызовах функции с параметрами разных типов и, в то же время, не страдает от низкой эффективности.

При анализе вызовов функций АДП работает с *мономорфными*, а не с *полиморфными* типами аргументов вызова. Под мономорфным типом понимается конкретный тип, с которым связано выражение по ходу выполнения программы. Примеры мономорфных типов: NoneType, int, класс A. Полиморфный тип для переменной или выражения определяется как совокупность всевозможных мономорфных типов для неё (него). Например, для следующего кода переменная x будет иметь полиморфный тип, равный {*NoneType, int, A*}:

```
x = None

...

x = 1

...

x = A()
```

Ключевую идею АДП можно проследить, если вернуться к аналогии между исполнением программы и её анализом. Во время исполнения программы все вызовы функций на самом деле осуществляются «мономорфно». Поясним, что имеется в виду. Представим себе функцию square, которая в программе принимает на вход аргумент a, имеющий полиморфный тип {*int, float*}.

```
def square(x):
    return x * x
a = 1
...
a = 3.14
...
b = square(a)
```

Очевидно, что во время каждого конкретного вызова функции square аргумент вызова может иметь либо тип int, либо тип float, но не оба сразу. Это наблюдение можно сформулировать следующим образом:

полиморфных вызовов функций не существует, о полиморфизме можно говорить лишь для выражений «вызов функции» в коде программы.

В АДП, в отличие от предшествующих алгоритмов, создаются лишь *мономорфные шаблоны*. Например, пусть имеется вызов max(x, y). Обозначим полиморфный тип аргумента x как X, тип аргумента y как Y. Предположим, мы каким-либо образом вычислили, что $X = \{x_1, x_2, \dots, x_s\}$, а $Y = \{y_1, y_2, \dots, y_t\}$. Чтобы вычислить тип для вызова max, АДП вычисляет декартово произведение типов аргументов вызова. В данном случае это множество пар: $X \times Y = \{(x_1, y_1), \dots, (x_1, y_t), \dots, (x_i, y_j), \dots, (x_s, y_1), \dots, (x_s, y_t)\}$. В общем случае это множество кортежей по n элементов в каждом, где n — число аргументов вызова.

Затем АДП для каждого $(x_i, y_j) \in X \times Y$ пропагирует типы аргументов по отдельному мономорфному шаблону — шаблону для этого кортежа. Если такой шаблон уже создавался ранее, он повторно используется (из него берутся возвращаемые типы), в противном случае создаётся новый шаблон, который в дальнейшем — для других вызовов max с аргументами (x_i, y_j) — будет доступен для повторного использования. Значение типовой переменной, связанной с вызовом функции, вычисляется как объединение множеств типов, возвращаемых в каждом используемом шаблоне. Ситуация проиллюстрирована на рис. 5.

В действительности при анализе некоторого вызова функции мы можем не знать точного полиморфного типа для каждого аргумента. Рассмотрим следующий код:

```
z = max(x, y)

...

y = 'abc'
```

В приведённом выше примере вызов функции max анализируется до того, как в множество возможных типов для y добавляется тип str. Это не является проблемой для АДП: декартово произведение вычисляется для **известных на текущий момент** типов аргументов вызова. Как только становятся известными новые возможные типы для аргументов, процедура вычисления повторяется заново. Декартово произведение — монотонная функция, поэтому при расширении типов аргументов «новое» декартово произведение включает в себя «старое» (см. рис. 6).

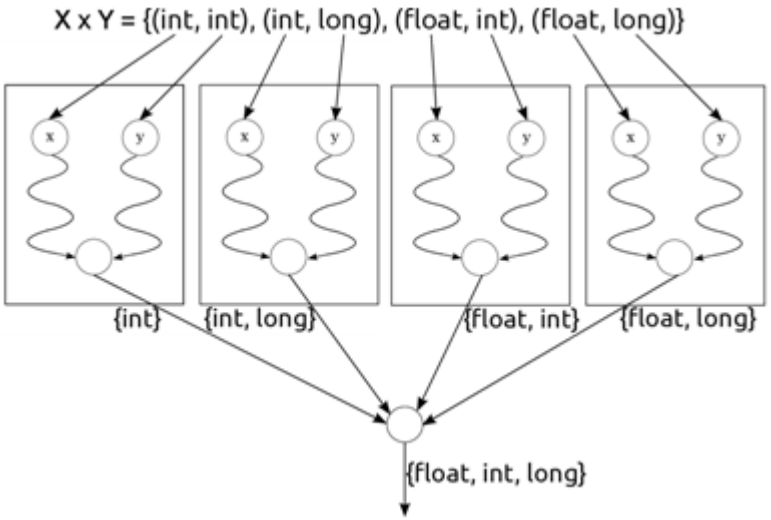

Рис. 5. АДП для вызова max(x, y), где X = type(x) ={int, float}, Y = type(y) = {int, long}.

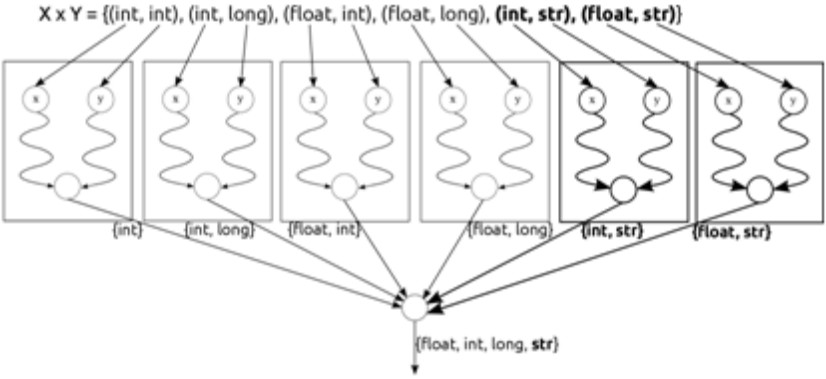

Рис. 6. АДП для вызова max(x, y), где X = type(x)= {int, float}, Y = type(y) = {int, long, str}. Здесь показано, что происходит при расширении типа аргумента вызова. Жирным шрифтом выделены «новые» структуры, создаваемые во время повторного вычисления декартова произведения.

Следовательно, мономорфные шаблоны не создаются напрасно. Объединение множеств также является монотонной функцией, поэтому тип вызова функции может только увеличиваться и т. д. Это гарантирует монотонность значения типовых переменных, т. е. типовые переменные всегда содержат правильные (но не обязательно все) возможные типы для определённого выражения.

В заключительной части описания АДП в [4] приводится оценка этого алгоритма:

- алгоритм является *точным* в том смысле, что он может анализировать цепочки вызовов полиморфных функций без потери точности;
- алгоритм является *эффективным*, потому что позволяет избежать избыточного анализа, так как вызов функции с аргументами мономорфных типов будет анализироваться ровно один раз.

АДП разрабатывался специально для языка программирования Self, который существенно отличается от языка Python, и непосредственное применение АДП к коду на Python не представляется возможным. В [6] приводится описание модифицированного АДП, который возможно использовать для анализа Python-программ. В частности, в новом алгоритме даются попытки решения проблем, с которыми АДП (для языка Self) не справляется либо справляется плохо. Вот эти проблемы:

- Алгоритм АДП хорошо справляется с полиморфическим полиморфизмом (см. выше), но плохо работает для *полиморфизма по данным*. Полиморфизм по данным — возможность хранения объектов разных типов в одной и той же «ячейке» данных. Под «ячейкой» понимается, например, элемент стандартной структуры «список» в языке Python. В списке [1, 3.14, 'abc'] разные «ячейки» хранят объекты, имеющие, соответственно, тип int, float и str.
- В АДП не описано, как обрабатывать внешний по отношению к анализируемой программе код. Например, метод append для CPython — стандартной реализации Python — написан на языке Си, и нужно иметь возможность анализировать такой метод с точки зрения вывода типов.
- АДП (в том виде, в котором он описан в [4]) зацикливается при наличии рекурсивных функций.

Несмотря на свои преимущества, алгоритм из [6] до настоящего момента реализован не был. Он служит основой для алгоритма вывода типов, реализованного в текущей работе. Однако, алгоритмы всё же существенно различаются, так как в исходный алгоритм было внесено значительное число изменений с целью решения практической задачи, сформулированной в начале работы: нахождения несоответствия типов в примере из проекта Gramps.

3. Описание реализации

Модуль вывода типов в настоящее время представляет собой отдельную программу Tirpan (**T**ype **i**nfe**r**ence-based **p**ython **an**alyzer), написанную на языке Python. Tirpan запускается на некотором Python-файле и по завершении своей работы сопоставляет каждое выражение в программном коде (исходного файла, а также транзитивно импортируемых из него файлов) с множеством возможных типов для

этого выражения. Мы исходим из естественного для статического анализа предположения, что программный текст всех файлов, нуждающихся в анализе, доступен. В остальном алгоритм вывода типов не накладывает никаких ограничений на анализируемые файлы. В этом отличие текущего алгоритма от исходного алгоритма. Исходный алгоритм, описанный в [6], предполагалось использовать для трансляции анализируемого файла на языке Python в код на Си, что требовало абсолютной точности вывода типов. Последнее требование, в свою очередь, не допускало наличия в коде динамических языковых конструкций вроде eval, exec и т. д. Для статического анализа стопроцентная точность вывода типов не требуется, поэтому подобные ограничения и отсутствуют.

При рассмотрении нашего алгоритма вывода типов мы будем придерживаться операционного подхода так же, как это делают авторы в [4, 6]. Это означает, что мы будем описывать алгоритм путём перечисления шагов, которые выполняются при анализе тех или иных элементов языка Python, а также структур данных, создаваемых во время выполнения этих шагов. Описание того, как происходит анализ различных конструкций Python, будет производиться от простых конструкций (например, стандартных структур, таких, как списки) к более сложным (например, классам и объектам). Кроме того, мы осветим процесс анализа внешнего по отношению к программе на языке Python кода.

3.1 Чувствительность к потоку выполнения

Текущий алгоритм, как и его предшественник, является нечувствительным к потоку выполнения (flow-insensitive). Это означает, что он добавляет возможные типы к переменным, опираясь только на исходный код анализируемой программы. Чувствительные к потоку выполнения (flow-sensitive) алгоритмы, напротив, добавляют возможные типы к конкретным *местам доступа к переменным*. Нечувствительные к потоку выполнения алгоритмы являются гораздо более простыми, чем чувствительные. В то же время первые теоретически являются менее точными, чем вторые. Действительно, рассмотрим следующий код:

```
def foo():
    if ...:
        x = 1
        return (x, square(x))
    else:
```

```
        x = 3.14
        return (x, square(x))
```

Переменной x в одной ветке условного оператора присваивается значение типа int, а в другой — типа float. Соответственно, функция foo вернёт либо пару значений типа int (из ветки if), либо пару значений типа float (из ветки else). Однако, текущий алгоритм, являясь нечувствительным к потоку выполнения, «склеит» два присваивания разных значений переменной x. В результате получится, что x имеет полиморфный тип {*int, float*}, а foo возвращает полиморфный тип, включающий в себя и {*tuple*(*int, float*)}, и {*tuple*(*float, int*)}, что неправильно.

Чувствительный к потоку выполнения алгоритм смог бы разделить два присваивания переменной x. Вообще говоря, текущий алгоритм можно было бы сделать чувствительным, если бы мы решили предварительно преобразовывать анализируемый исходный код в т. н. SSA-представление (static single assignment form) [7]. В этом представлении переменные переименовываются таким образом, чтобы чтение некоторой переменной соответствовало записи в неё. В таком случае в выше приведённом примере переменная x в одной ветке была бы переименована в x1, а в другой — в x2, и коллизии бы не произошло. Однако и этот, и некоторые другие методы всё равно помогают лишь в простых случаях, когда одной переменной в одной и той же области видимости присваиваются разные значения. В более сложных случаях не спасают и они. Рассмотрим следующий пример кода на языке Python:

```
def foo(x):
    if isinstance(x, int):
        return x
    else:
        return None
y = foo(1)
```

Чувствительный к потоку выполнения алгоритм определил бы, что тип выражения foo(1) (и, следовательно, переменной y) равен {*int*}. Поскольку нечувствительный к потоку выполнения алгоритм не учитывает условия в if, он выведет для y неточный тип, равный {*int, NoneType*}.

В то же время следует отметить, что присваивать одной переменной в одной области видимости значения разных типов — плохая практика. Таким образом, получается, что сильно усложнять алгоритм пришлось

бы ради лучшей поддержки кода не очень высокого качества. Поэтому было решено отказаться от такого усложнения и оставить алгоритм нечувствительным к потоку выполнения программы.

3.2 Общая схема работы программы

Общая схема работы программы изображена на рис. 7. Программе в виде аргументов командной строки передаётся путь к файлу, который необходимо проанализировать, а также, возможно, некоторые флаги. В модуле Tirpan происходит обработка этих флагов. Если всё прошло успешно, в модуль Importer передаётся команда «импорт главного модуля (модуля __main__)» с указанием адреса исходного файла. Importer считывает этот файл и передаёт его содержимое в модуль Parser, где по исходному коду строится дерево абстрактного синтаксиса (abstract syntax tree, AST). Для этого используются стандартные средства языка Python (модуль ast, включающий в себя функцию ast.parse) [8]. Сгенерированный AST пересылается обратно в Importer. Внутри себя Importer содержит *таблицу модулей*, каждая запись в которой содержит сведения об отдельном модуле: информацию о пути, по которому был найден импортированный файл, ссылку на AST для этого файла и некоторые другие признаки. Таблица модулей заполняется первой записью для главного модуля, после чего AST передаётся в модуль TIVisitor (type inference visitor). TIVisitor фактически представляет собой обходчик синтаксического дерева, в котором на определённые узлы дерева «навешивается» дополнительная информация о возможных типах для этих узлов. Понятно, что в процессе обхода дерева могут встретиться import-директивы. В таком случае модулю Importer пересылается команда «импорт модуля x», и (если модуль не был импортирован ранее) вновь выполняется процедура генерации синтаксического дерева, создания записи в таблице модулей и обхода полученного дерева.

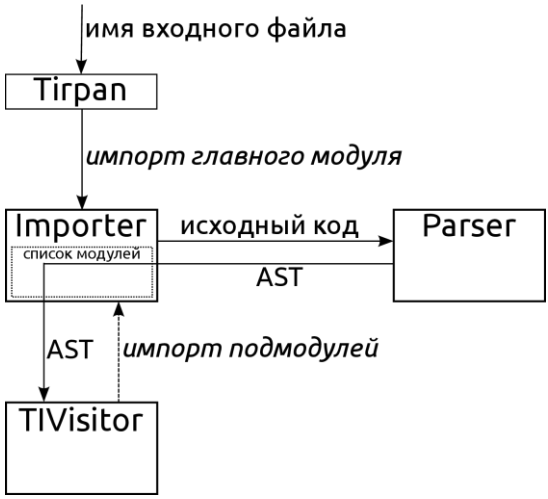

Рис. 7. Общая схема работы программы.

3.3 Структуры данных

В программе Tirpan используются две основные иерархии классов:

- классы, унаследованные от TypeNode (см. рис. 8), используются для представления типов, которые пропагируются по графу типовых переменных. Некоторые классы из этой иерархии имеют атрибуты, необходимые для правильного представления типа. Так, например, у классов TypeList и TypeTuple есть атрибут elems, хранящий множество типов элементов списка или, соответственно, кортежа. У классов TypeInt, TypeStr и TypeUnicode есть атрибут value, который хранит конкретное значение (например, 1, 'abc' или u'abc'), если оно может быть вычислено, или None в противном случае. Это было сделано из-за того, что для точного анализа программы часто недостаточно просто знать, что определённая переменная хранит в себе некие строковые или целочисленные значения, а требуется точно вычислять это значение. В текущей реализации было решено отказаться от хранения значений для других встроенных типов. Это было сделано с целью улучшения производительности и экономии памяти.
- классы, унаследованные от TypeGraphNode, используются для представления типовых переменных, из которых конструируется граф вывода типов. При обходе синтаксического дерева для различных языковых конструкций создаются объекты различных классов этой иерархии, а «навешивание» информации о типах на AST заключается в том, что к узлам дерева динамически

добавляется атрибут link, ссылающийся на соответствующую типовую переменную. У всех классов TypeGraphNode-иерархии есть атрибут nodeType, хранящий в себе множество типов, т. е. экземпляров классов TypeNode-иерархии. Также некоторые классы могут иметь специфичные для них атрибуты: например, у VarTypeGraphNode есть поле name (имя переменной), а у AttributeTypeGraphNode — поле objects (множество всевозможных объектов для выражения вида x.y) и т. д.

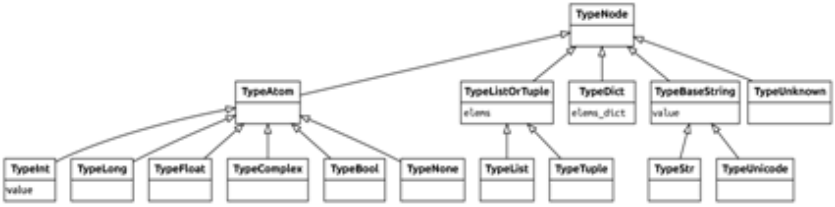

Рис. 8. Иерархия классов, унаследованных от TypeNode.

Вне двух основных иерархий находится класс Scope, используемый для представления областей видимости имён. В Scope хранится словарь, где ключами являются строки (имена переменных), а значениями — соответствующие переменные (объекты класса VarTypeGraphNode). Кроме того, у класса Scope есть атрибут parent, хранящий в себе ссылку на более верхнюю (родительскую) область видимости. У переменных также есть поле parent — оно указывает на область видимости, в которой находится данная переменная.

3.4 Константы и стандартные структуры языка Python

Самый простой случай при выводе типов — анализ констант (целочисленных, вещественных и т. д.) и строковых литералов. Каждый узел AST, соответствующий константе (литералу), связываются со своей типовой переменной — объектом класса ConstTypeGraphNode. Значением атрибута nodeType у этой типовой переменной становится множество из одного элемента: экземпляра класса из TypeNode-иерархии, соответствующего типу константы. Например, для константы 3.14 таким элементом является объект класса TypeFloat, а для строкового литера 'abc' — объект класса TypeStr со значением 'abc' в поле value (как уже говорилось, для целых чисел и строк при анализе учитывается не только тип, но и значение).

Теперь рассмотрим, как происходит анализ стандартных структур языка Python: списков, кортежей и словарей. Если при обходе синтаксического дерева встретился узел «список» (ast.List), то:

1. Создаётся типовая переменная (назовём её list_var) — объект класса ListTypeGraphNode.
2. Обходятся поддеревья, соответствующие элементам списка.
3. От каждой типовой переменной (назовём её elem_var), соответствующей i-му элементу списка, проставляется связь вида Elem к переменной list_var.

При добавлении некоторого ребра (связи) между типовыми переменными автоматически происходит пропагация типов вдоль этого ребра. То, как в результате пропагации изменится значение типовой переменной на конце ребра, зависит от вида связи и, возможно, от того, какому классу принадлежит эта типовая переменная. Если значение переменной меняется, то изменение распространяется по рёбрам, исходящим из этой переменной, и т. д.

Связь вида Elem от elem_var к list_var означает, что в списки, входящие в множество значений list_var, необходимо добавить типы, пришедшие из elem_var. Ситуация проиллюстрирована на рис. 9. Для представления типов «списки» используются объекты класса TypeList.

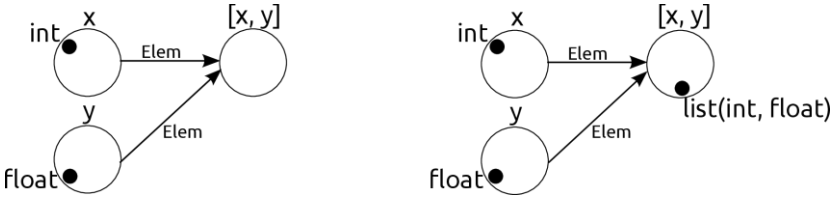

Рис. 9. Связи вида Elem, созданные при анализе списка [x, y]. Слева показаны типовые переменные до пропагации типов по соответствующим рёбрам, справа — после. Если в type(x) добавится новый тип, например, long, тип на другом конце связи превратится из {list(int, float)} в {list(int, long, float)}.

Аналогичным образом при анализе кортежей создаются и обрабатываются типовые переменные класса TupleTypeGraphNode. В отличие от ситуации со списками, связь Elem от elem_var к tuple_var задаётся с аргументом — индексом элемента в кортеже. Это необходимо для правильного вычисления возможных типов tuple_var. А при анализе словарей используются типовые переменные класса DictTypeGraphNode. К таким переменным могут проставляться связи двух видов: вида Key (от ключей словаря) и вида Val (от значений словаря).

3.5 Имена

Если при обходе дерева встретился узел «имя» (ast.Name), производится поиск типовой переменной класса VarTypeGraphNode с соответствующим именем (поле node.name в узле). Если такая переменная найдена, к ней проставляется ссылка от узла. В противном случае переменная с нужным именем создаётся в текущей области

видимости и опять же «навешивается» на анализируемый узел. Из этой простой логики есть свои исключения: например, если в теле некой функции в левой части присваивания есть переменная x, определять, глобальная или локальная переменная имеется в виду, необходимо по наличию директивы global x в том же теле функции. Такая семантика учитывается при анализе ast.Name.

3.6 Присваивания

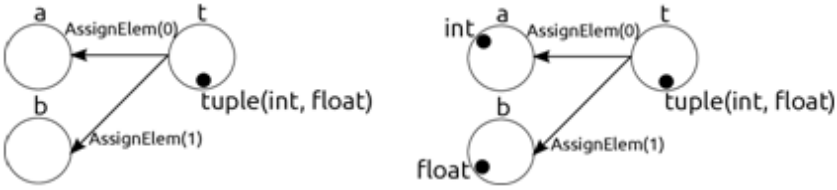

Рис. 10. Связи вида AssignElem, создаваемые при анализе присваивания списку переменных, например: a, b = t. Слева показаны типовые переменные до пропагации типов по соответствующим рёбрам, справа — после.

При анализе обычных присваиваний вида x = y сначала обходятся левая и правая часть присваивания. Затем от типовой переменной, соответствующей правой части, проставляется связь вида Assign к типовой переменной, соответствующей левой части. Связи вида Assign соответствуют обычным связям из «базового» алгоритма: все типы, которые были выведены для типовой переменной y, автоматически пропагируются в типовую переменную x.

Помимо обычных присваиваний, в Python присутствует конструкция вида x_1, …, x_n = y (присваивание списку переменных). При анализе таких присваиваний в первую очередь обходятся правая часть присваивания и все переменные в левой части. Затем от типовой переменной, соответствующей правой части, проставляются связи вида AssignElem к типовым переменным, соответствующим элементам списка присваивания. Для правильного вывода типов при добавлении связи к ней приписывается целочисленный аргумент: индекс переменной в списке. Ситуация проиллюстрирована на рис. 10. Связь вида AssignElem также используется при обработке конструкции in, в которой тип итератора определяется как множество элементов, содержащихся в типах коллекции. Так, для следующего примера будет вычислено, что тип коллекции равен {*list*(*int, float, str*)}, а тип итератора, соответственно, равен {*int, float, str*}:

```
collection = [1, 3.14, 'abc']
for iterator in collection:
    print iterator
```

3.7 Функции

В программе на языке Python объявление именованной функции, например foo, равносильно объявлению безымянной функции и связыванию её с переменной foo (в дальнейшем с foo может быть связано значение любого другого типа). По аналогии с выполнением программы при анализе объявления функции создаются два узла из TypeGraphNode-иерархии: UsualFuncDefTypeGraphNode и UsualVarTypeGraphNode (с таким же именем, как у функции), причём от первого объекта ко второму проставляется связь вида Assign. В объекте класса UsualFuncDefTypeGraphNode хранятся ссылки на параметры функции (в виде области видимости, в которую были добавлены соответствующие переменные) и на AST для тела функции. Этот AST при анализе определения функции не обходится — его обход откладывается до момента вызова, что также соответствует семантике языка Python. Кроме того, в объекте UsualFuncDefTypeGraphNode создаётся изначально пустое множество шаблонов для функции. Определения безымянных (лямбда-) функций обрабатываются аналогично, только в этом случае элемента UsualVarTypeGraphNode не создаётся.

При обработке узла «вызов функции» в первую очередь обходится поддерево «имя функции», затем поддеревья, соответствующие аргументам вызова, и в конце концов создаётся объект класса FuncCallTypeGraphNode, который «навешивается» на узел. К этому объекту проставляется два вида связей:

- от типовых переменных, соответствующих аргументам вызова, — связи вида Arg;
- от типовой переменной, соответствующей имени вызываемой функции, — связь вида Func.

Далее обработка вызова происходит в соответствии с АДП. Сначала вычисляется декартово произведение типов аргументов вызова и возможных значений типовой переменной, соответствующей имени функции. Затем для каждого элемента декартова произведения — обозначим элемент как $(f, a_1, \dots, a_n)$ — проверяется соответствие числа параметров f и аргументов вызова (и в параметрах, и в аргументах могут встречаться *args- и **kwargs-конструкции), т. е. возможен ли вообще вызов f с аргументами $(a_1, \dots, a_n)$. Если вызов возможен, производится проверка, не существует ли у функции шаблона для данных аргументов. Если шаблон существует, то возможные типы возвращаемого значения функции будут браться из него. Если же шаблона не существует, то он создаётся в виде структуры, хранящей следующие атрибуты:

- args — типы аргументов, для которых был создан шаблон;
- ast — AST для шаблона;

- result — множество возвращаемых из функции типов (инициализируется пустым множеством).

Каждый раз при создании нового шаблона из соответствующего объекта UsualFuncDefTypeGraphNode берётся сохранённый AST для функции и генерируется его точная копия, которая записывается в атрибут ast в шаблоне. Затем делается точная копия параметров функции, и в скопированные типовые переменные записываются мономорфные типы аргументов вызова. После этого производится обход скопированного синтаксического дерева для функции. Работа именно с различными копиями одного и того же дерева нужна для того, чтобы не смешивать типы параметров, локальных переменных и, в конечном счёте, возвращаемых значений функций для разных шаблонов. Если при обходе скопированного дерева встречаются узлы «оператор return», типы возвращаемых в операторе значений добавляются в множество, которое хранится в атрибуте result в шаблоне. После обработки шаблона типы из result добавляются в поле nodeType у объекта FuncCallTypeGraphNode, т. е. в значение соответствующей вызову типовой переменной.

Благодаря тому, что шаблон для функции создаётся перед обходом её тела, зацикливания при анализе рекурсивных функций не возникает. Действительно, рассмотрим рекурсивную реализацию функции factorial:

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
```

При обходе выражения factorial(n - 1) алгоритм обнаруживает, что выражение n - 1 имеет тип int и что соответствующий шаблон для factorial уже был создан. Повторного обхода функции, который привёл бы к зацикливанию, не происходит.

Наличие связей вида Arg и Func гарантирует, что при расширении типов аргументов вызова или вызываемых функций происходит повторное вычисление декартова произведения и, возможно, расширение типа выражения «вызов функции».

3.8 Классы и объекты

Семантика определения класса в Python в чём-то схожа с семантикой определения функции. Определение класса A предписывает создать безымянный класс и связать его с переменной A. Однако, в отличие от

ситуации с функцией, тело класса начинает выполняться (интерпретироваться) сразу же после связывания. Поведение алгоритма вывода типов моделирует эту семантику. При анализе узла «определение класса» создаются объекты UsualClassDefTypeGraphNode и UsualVarTypeGraphNode (с именем класса), и, также как для функций, от первого ко второму объекту проставляется связь вида Assign. Объект класса UsualClassDefTypeGraphNode хранит в себе следующую информацию:

- данные о базовых классах для текущего класса (в настоящий момент не учитывается, что базовые классы могут быть изменены динамически);
- область видимости (изначально пустая), в которую будут добавляться все переменные, являющиеся членами текущего класса;
- список экземпляров класса (изначально пустой).

После того, как были созданы типовые переменные, текущей областью видимости становится область видимости класса и производится обход его тела.

Создание экземпляра определённого класса A в языке Python выглядит как вызов функции A с некоторым (возможно, пустым) списком аргументов, которые передаются в конструктор класса A. Такая семантика учитывается при анализе вызовов функций. Если оказывается, что в некотором элементе декартова произведения f (см. 3.7) — это определённый класс A, то выполняются специальные действия, несколько отличающиеся от обычной обработки вызова функции:

1. Производится поиск конструктора A.__init__. Если конструктора в классе A нет, считается, что происходит вызов конструктора по умолчанию, принимающего пустой список аргументов.
2. Проверяется, соответствуют ли параметры конструктора аргументам вызова, т. е. возможен ли вызов конструктора с такими аргументами.
3. Если проверка соответствия параметров и аргументов прошла успешно, ищется шаблон, соответствующий аргументам вызова. Если такой шаблон найден, то тип возвращаемого значения берётся из него.
4. В противном случае создаётся новый объект ClassInstanceTypeGraphNode, хранящий в себе информацию о созданном в результате вызова экземпляре класса (в текущей реализации и ClassDefTypeGraphNode, и ClassInstanceTypeGraphNode, хотя и являются частью TypeGraphNode-иерархии, используются также в качестве типа, как если бы они был частью TypeNode-иерархии). Созданный объект добавляется в список экземпляров класса A, хранящийся в

соответствующем объекте ClassDefTypeGraphNode. Проставляется и ссылка от поля cls в ClassInstanceTypeGraphNode к соответствующему ClassDefTypeGraphNode.

Дальнейшие действия практически совпадают с обработкой обычного вызова функции: производится копирование AST и параметров функции, связывание параметров с мономорфными типами аргументов и обход AST. Единственным отличие от обычного вызова здесь является то, что параметр self в конструкторе связывается с только что созданным объектом ClassInstanceTypeGraphNode.

3.9 Атрибуты и операции взятия элемента коллекции

Поскольку работа с экземплярами класса осуществляется при помощи операции чтения или записи атрибутов (это верно как для полей, так и для методов в объектах), необходима поддержка соответствующей конструкции при выводе типов.

Если при обходе дерева встретился узел «доступ к атрибуту», то в первую очередь обходится поддерево value, представляющее объект, у которого берётся атрибут, а затем создаётся типовая переменная — объект класса AttributeTypeGraphNode (назовём её var_attr). В эту типовую переменную записывается информация об имени атрибута, необходимая для правильного определения типа выражения. От типовой переменной, «навешенной» на поддерево value (назовём переменную var_value), проставляется связь вида AttrObject к var_attr. В обратную сторону (от var_attr к var_value) также проставляется связь, но вида AssignObject. Двухсторонние связи отражают тот факт, что при расширении типа объектов необходимо пересчитывать тип атрибутов, и наоборот.

К связям AttrObject/AssignObject могут прибавляться связи, созданные во время анализа присваиваний. Так, для самого простого случая записи в атрибут (выражения x.y = z) к var_attr будет идти связь вида Assign от типовой переменной для z. При пропагации новых типов по связи Assign в типовой переменной var_attr происходит процедура пересчёта значения переменной: сначала в функции set_attributes записывается информация о том, что атрибут self.attr во всех возможных объектах (это множество хранится в self.objects) может принимать новые типы — элементы множества self.values, затем в функции get_attributes вычисляется новое значение типовой переменной (поле self.nodeType):

```
def process(self):
    set_attributes(self.objects, self.attr,
self.values)
```

```
    self.nodeType =
get_attributes(self.objects, self.attr)
```

В функции get_attributes учитывается, что некоторые объекты могут иметь метод __getattr__, переопределяющий стандартную операцию чтения атрибута (например, в методе может быть реализовано чтение одноимённого атрибута из другого объекта).

Почти аналогично производится вывод типов для узлов «взятие элемента» (ast.Subscript), например, для выражения x[0]. Для таких узлов создаётся типовая переменная класса SubscriptTypeGraphNode, хранящее в себе вместо имени атрибута информацию о типе (и, возможно, значении) выражения «индекс». Как и в случае с атрибутами, здесь генерируются связи вида AttrObject и вида AssignObject. Метод process в SubscriptTypeGraphNode похож на аналогичный метод в AttributeTypeGraphNode, только в первом вызываются функции set_subscripts и get_subscripts.

3.10 Внешний код

Для поддержки написанного не на языке Python кода в TypeGraphNode-иерархию были добавлены новые классы: ExternVarTypeGraphNode и ExternFuncDefTypeGraphNode. Как следует из названия, они нужны для моделирования внешних переменных и, соответственно, внешних функций. Расскажем подробнее об каждой из групп.

Примером внешней (по отношению к коду на Python) переменной является path из модуля sys. Перед началом выполнения Python-программы sys.path связывается со списком строк, каждая из которых представляет собой некоторый путь в файловой системе. Список частично формируется из содержимого переменной окружения PYTHONPATH, но содержит в себе и элементы, зависящие от текущей инсталляции интерпретатора Python. Для моделирования подобных внешних переменных перед непосредственным запуском вывода типов (импортом модуля __main__) происходит создание ряда объектов ExternVarTypeGraphNode, в том числе и для sys.path. Для каждого объекта указана определённая функция. После создания некоторого объекта класса ExternVarTypeGraphNode вызывается соответствующая ему функция, и результат её работы записывается в поле nodeType у объекта. Например, для sys.path функция инициализации выглядит следующим образом:

```
def quasi_sys_path():
    res = TypeList()
```

```
    tmp = []
    ...
    for elem in sys.path[1:]:
        tmp.append(get_new_string(elem))
    res.elems = tmp
    return res
```

В качестве примера внешней функции можно привести уже упоминавшуюся append, принимающую на вход объект-список и элемент, который в этот список необходимо добавить. Для обработки append при выводе типов ей, как и в случае с sys.path, ставится в соответствие некоторая функция, только на этот раз функция принимает на вход типы параметров для append. На выходе по ним строится тот тип, который имело бы возвращаемое значение append, а также, возможно, моделируется внесение изменений в типы входных параметров (в случае с append тип первого параметра меняется в ходе выполнения функции). Таким образом работа функции append моделируется с точки зрения типов на входе в функцию и на выходе из неё:

```
def quasi_append(scope):
    type1 = getParamType(scope, 1)
    type2 = getParamType(scope, 2)
    if not isinstance(type1, TypeList):
        return set([type_none])
    type1.add_elem(type2)
    return set([type_none])
```

Каждой поддерживаемой внешней функции ставится в соответствие типовая переменная класса ExternFuncDefTypeGraphNode. У объектов этого класса нет атрибута ast, зато в атрибуте quasi хранится соответствующая моделирующая функция. Если при обработке вызова функции встретился объект ExternFuncDefTypeGraphNode, вместо обхода сохранённого AST происходит вызов соответствующей моделирующей функции.

4. Результаты

Как уже говорилось, для демонстрации работоспособности Tirpan используется пример несоответствия типов из проекта Gramps. В этом примере переменная css_filename в результате работы программы может принимать значение [], передача которого в os.path.basename вызывает ошибку:

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

Тестирование Tirpan показало, что тип list() (пустой список) правильно выводится в качестве одного из возможных типов для css_filename. Однако, информация о типах, «навешенных» на узлы синтаксического дерева, является внутренней и пользователю непосредственно не видна. Поэтому было решено реализовать прототип обнаружителя дефектов (чекера), который выводил бы сообщение о дефекте «передача аргумента, не являющегося строкой, в функцию os.path.basename» в случае наличия такого дефекта в коде.

В текущей реализации чекер работает не после этапа вывода типов, а непосредственно во время него. Он представляет собой callback (функцию обратного вызова), который вызывается каждый раз при обходе нового узла AST. Чекер проверяет, не встретилась ли функция os.path.basename и, если встретилась, каковы возможные типы её (первого) аргумента. Если не все типы являются строками (т. е. объектами классов TypeStr и TypeUnicode), чекер рапортует сообщение о дефекте несоответствия типов.

Для того, чтобы подобное сообщение было более информативно, в модуль вывода типов было добавлено сохранение трассировки стека функций (traceback). Этой цели в Tirpan служит объект класса Backtrace, хранящий в себе стек *фреймов* — элементов **Ошибка!**, входящих в декартово произведение и анализируемых при обработке вызовов функций. Перед обходом AST для некоторого шаблона функции соответствующий фрейм добавляется в стек, а после обхода — удаляется из стека. Сообщение о найденном дефекте, содержащее трассировку стека, выводится по окончании исполнения программы и выглядит следующим образом:

```
FUNC.ARG.WRONG: 'os.path.basename' expects
<basestring object>, not []. Backtrace:
    <HtmlDoc object>.set_css_filename([])
    <TextReportDialog object>.make_document()
    <TextReportDialog object>.on_ok_clicked(None)
    <Dialog object>.run()
    report(<DbState object>, <DisplayState
object>, ?,
           <class 'FamilySheet'>, <class
'FamilySheetOptions'>,
           'Family Sheet', 'FamilySheet',
           <int value>, <bool value>)
    run_plugin(<PluginData object>, <DbState
object>, <DisplayState object>)
    <lambda>(None)
    main()
    startgtkloop([], <ArgParser object>)
    run()
```

Однако, предположим, что код функции set_css_filename был исправлен, чтобы избежать несоответствия типов, например, следующим образом (очевидно, что возможны и другие варианты):

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

Здесь в os.path.basename уже не может передаваться пустой список, так как он вычисляется в False. Однако, поскольку алгоритм вывода типов, лежащий в основе Tirpan, нечувствителен к потоку выполнения, изменение условия будет проигнорировано и дефект FUNC.ARG.WRONG будет по-прежнему рапортоваться. Чтобы подобного ложного срабатывания (false positive) не происходило, в алгоритм вывода типов было внесено небольшое исправление, позволяющее фильтровать типы для переменных, используемых в условиях if.

Рис. 11. Связь вида AssignTrue, создаваемая при анализе условия if x:. Слева показаны типовые переменные до пропагации типов по соответствующим рёбрам, справа — после

Исправление такое: если в условии if встречается конъюнкция (возможно, вырожденная), то её операнды обходятся слева направо, и для каждого имени (узла ast.Name) выполняется следующая последовательность действий. Пусть мы встретили имя x. В этом случае создаются новая область видимости и новая же переменная с именем x, которая добавляется в созданную область видимости (чтобы не путать новую переменную со старой, будем называть новую x'). Затем между ними проставляются следующие связи: от x' к x обычная связь вида Assign, а от x к x' специальная связь вида AssignTrue, по которой могут распространяться лишь типы, вычисляемые в True. Если внутри ветки if с x будет связано значение нового типа, этот тип будет добавлен в x' и автоматически распространится в x. Связи между типовыми переменными проиллюстрированы на рис. 11.

Описанная модификация алгоритма позволяет избежать ложного срабатывания как для приведённого выше примера, так и, например, для варианта, когда старый условный оператор (if os.path.basename(css_filename):) добавляется в тело нового условного оператора (if css_filename:).

Тем не менее на проекте Gramps было обнаружено 4 ложных срабатывания реализованного чекера. Исследование показало, что ложные срабатывания вызваны следующими причинами:

- не всегда корректной работой со значениями словарей, из-за чего в множество возможных типов для аргумента os.path.basename попадают неправильные элементы;

- недостаточной чувствительностью к потоку выполнения: тот факт, что «плохие» типы отсеиваются с помощью условий в операторе if, часто не учитывается при выводе типов.

Второй пункт требует пояснений: хотя в алгоритм и была внесена модификация, учитывающая использование имён в условиях if, в более сложных случаях фильтрация типов не работает. Например, если вместо имени в условии стоит атрибут некоторого объекта или вызов функции. Вот один из примеров реального кода (стандартный модуль webbrowser), для которого в текущей реализации заданные в операторе if условия игнорируются:

```
if isinstance(name, basestring):
    self.name = name
else:
    self.name = name[0]
self.basename = os.path.basename(self.name)
```

Перечислим другие известные проблемы и ограничения в текущей реализации Tirpan:

- существуют проблемы с производительностью (время работы — несколько десятков минут) и объёмом потребляемой оперативной памяти (около 3 ГБ) на проекте Gramps;
- для того, чтобы избежать зацикливания алгоритма при наличии циклических связей (создаваемых, например, при анализе кода x = [x]) число возможных типов для любого выражения ограничено сверху константой, что может ухудшать точность анализа;
- с целью улучшения производительности и экономии памяти введены ограничения на количество возможных шаблонов для одной функции; кроме того, не для всех файлов вычисляются значения (а не только типы) целочисленных и строковых констант, что также может ухудшать точность.

5. Заключение

На основе модифицированного алгоритма декартова произведения был реализован работоспособный модуль вывода типов, который в состоянии обнаруживать несоответствие типов на реальных Python-проектах. Информация, полученная на этапе вывода типов, может быть использована не только для поиска дефектов, но и для рефакторинга кода, навигации по нему или вычисления определённых метрик.

Представляются возможными несколько направлений дальнейшей работы:

- устранение проблем производительности и вытекающих из них ограничений;
- улучшение вывода типов для уменьшения числа ложных сообщений о дефектах;
- генерация для каждого обнаруженного дефекта его трассы — последовательности шагов, из которой можно понять, как неправильный тип попал в функцию, содержащую несоответствие типов (что должно помочь пользователю статического анализатора определить, является дефект истинным или ложным).

Исходя из того, что трассировка стека, которая в текущей реализации выводится вместо трассы дефекта, является малоинформативной, последнее направление деятельности видится весьма актуальным. Вероятно, при выводе типов наряду с обычными связями следует генерировать «обратные» им, чтобы впоследствии проследить, откуда «плохие» типы пришли в место, где располагается дефект. В таком случае генерация трассы будет представлять собой поиск определённого пути в графе типовых переменных.

6. Список литературы

[1]. В. Савицкий, Д. Сидоров. Инкрементальный анализ исходного кода на языках C/C++. Труды ИСП РАН, том 22, сс. 119—129, 2012.

[2]. TIOBE Programming Community Index for April 2013. http://tinyurl.com/cgjbmjc

[3]. Gramps Bug Report 005023. http://www.gramps-project.org/bugs/view.php?id=5023

[4]. O. Agesen. The Cartesian Product Algorithm. ECOOP'95 Proceedings of the 9th European Conference on Object-Oriented Programming (1995).

[5]. J. Palsberg, M. Schwartzbach. Object-Oriented Type Inference. In OOPSLA'91 Object-Oriented Programming Systems, Languages and Applications, pp. 146—161, Phoenix, Arizona, Oct. 1991.

[6]. M. Salib. Starkiller: a static type inferencer and compiler for Python. The Master of Engineering degree thesis. Massachusetts Institute of Technology, 2004.

[7]. B. Alpern, M. Wegman, K. Zadeck. Detecting equality of values in programs. In Conference Record of the 15th ACM Symposium on Principles of Programming Languages (Jan. 1988), ACM, New York, pp. 1—11.

[8]. Abstract Syntax Trees: ast module. http://docs.python.org/2/library/ast.html

Type inference for Python programming language

Bronshteyn I. E. (ISP RAS, Moscow, Russia)
ibronstein@ispras.ru

Abstract. The article presents type inference for programs written in Python programming language. At first, type inference algorithms for parametric polymorphism that were described in the scientific literature are reviewed. The algorithms evolved from “basic” algorithm (also known as Palsberg — Schwartzbach algorithm) to Cartesian product algorithm (CPA). It was shown that CPA is both precise and efficient, however it has to be modified to be used on Python code. Afterwards, we present a new algorithm (a modification of CPA) for Python. It is shown how type inference module using the new algorithm analyses various Python language structures: constants (literals), basic Python collections, variable names, assignments, function and class definitions, attribute and index expressions. It is also shown how the algorithm deals with external (non-Python) functions using special annotations that specify output types depending on input types of the function. Afterwards, the results of work on the prototype (module that implements described type inference algorithm) are presented. The paper is concluded by an overview of possible future work directions such as generating a defect trace, i.e. description that specifies how expression got its incorrect types.

Keywords: python; type inference; dynamic typing; static analysis; defects detection.

References

[1]. V. Savitskij, D. 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).

[2]. TIOBE Programming Community Index for April 2013. http://tinyurl.com/cgjbmjc

[3]. Gramps Bug Report 005023. http://www.gramps-project.org/bugs/view.php?id=5023

[4]. O. Agesen. The Cartesian Product Algorithm. ECOOP’95 Proceedings of the 9th European Conference on Object-Oriented Programming (1995).

[5]. J. Palsberg, M. Schwartzbach. Object-Oriented Type Inference. In OOPSLA’91 Object-Oriented Programming Systems, Languages and Applications, pp. 146—161, Phoenix, Arizona, Oct. 1991.

[6]. M. Salib. Starkiller: a static type inferencer and compiler for Python. The Master of Engineering degree thesis. Massachusetts Institute of Technology, 2004.

[7]. B. Alpern, M. Wegman, K. Zadeck. Detecting equality of values in programs. In Conference Record of the 15th ACM Symposium on Principles of Programming Languages (Jan. 1988), ACM, New York, pp. 1—11.

[8]. Abstract Syntax Trees: ast module. http://docs.python.org/2/library/ast.html