Санкт-Петербургский государственный университет

***Мирошников Владислав Игоревич***

Выпускная квалификационная работа

# Статический вывод типов для языка Python в интегрированной среде разработки

Уровень образования: бакалавриат

Направление *09.03.04 «Программная инженерия»*

Основная образовательная программа *СВ.5080.2019 «Программная инженерия»*

Научный руководитель:
доцент кафедры СП, к.ф.-м.н. Д. Ю. Булычев

Консультант:
инженер ключевых проектов ООО «МПГ Айти Солюшнз» А. А. Захаров

Рецензент:
ведущий инженер ключевых проектов ООО «Техкомпания Хуавэй» О. Е. Лукьянова

Санкт-Петербург
2023

Saint Petersburg State University

***Vladislav Miroshnikov***

Bachelor's Thesis

# Static Type Inference for Python in IDE

Education level: bachelor

Speciality *09.03.04 «Software Engineering»*

Programme *CB.5080.2019 «Software Engineering»*

Scientific supervisor:
System Programming chair docent, C. Sc. D.Y. Bulychev

Consultant:
Senior engineer at «MPG IT Solutions LLC» A.A. Zakharov

Reviewer:
Principal engineer at «Huawei Technologies CO.LTD» O.E. Lukianova

Saint Petersburg
2023

# Оглавление

# Введение

В наши дни сложно представить разработку программного обеспечения без использования дополнительных средств, в частности интегрированных сред разработки (IDE), значительно упрощающих процесс разработки и предоставляющих большой спектр различной функциональности, например, технологии автодополнения, анализа кода, системы рефакторингов и многое другое. В качестве наиболее популярных сред разработки можно привести среду `IntelliJ IDEA` [1] от компании JetBrains или же `Microsoft Visual Studio` [2].

Неотъемлемой частью в любой среде разработки является ее кодовая модель (Code Model). На ее основе уже строится вся остальная функциональность. Компонента кодовой модели отвечает в том числе и за синтаксический анализ исходного кода языка, построение синтаксического и расширенного семантического дерева, построение различных индексов и специальных код-модельных символов.

В крупной телекоммуникационной компании было принято решение разработать собственную интегрированную среду разработки, именуемую SRC IDE (Saint Petersburg Research Center IDE)[1]. SRC IDE поддерживает язык программирования Java, а также на текущий момент идет активная разработка и добавление поддержки языка Python в силу его высокой и растущей популярности [3]. При этом стоит понимать, что SRC IDE проектируется и разрабатывается с учетом мультиязыковой составлющей: компоненты и подсистемы IDE должны обеспечивать внедрение и других языков или семейств языков.

Одной из ключевых и важных задач в подсистеме кодовой модели является вывод типов для выражений соответствующего языка, поскольку на его основе осуществляется разработка другой функциональности IDE, например, разрешения квалифицированных имен в коде и впоследствии поддержка навигации, технологии автодополнения, анализ корректности семантики до непосредственного запуска программы. Также нужно принимать во внимание тот факт, что задача вывода ти-

[1]Название изменено с целью соблюдения соглашения о неразглашении.

пов решается для языков с различными системами типизаций: статической и динамической. Однако для IDE особенно важно выводить типы статически, так как анализ кода для пользователя должен происходить до запуска самой программы.

Стоит учитывать, что язык Python имеет систему со строгой динамической типизацией, где типы определяются непосредственно в процессе работы программы, а также обладает свойством «утиной» типизации, определяющим, что атрибуты объекта в определенных случаях могут быть важнее типа самого объекта. Принимая во внимание требования со стороны IDE, в данной работе предлагается реализация статического вывода типов для языка Python с учетом его особенностей и интеграция в SRC IDE. Учитывая, что в рамках данной работы основной акцент делается на языке Python, далее вместо мультиязыковой SRC IDE будет рассматриваться Python IDE.

# 1. Постановка задачи

Целью выпускной квалификационной работы является реализация статического вывода типов для языка Python в рамках добавления поддержки Python в SRC IDE.

Для достижения цели были поставлены следующие задачи:

1. Изучить систему типов языка Python, а также существующие целевые решения, позволяющие статически выводить типы.

2. Рассмотреть различные подходы и алгоритмы вывода типов для языка Python, а также выбрать наиболее подходящий для дальнейшей реализации с учетом выявленных требований и ограничений.

3. На основе выбранного подхода спроектировать систему типов в рамках Python IDE, учитывающую особенности и свойства типовой системы Python.

4. Реализовать подсистему статического вывода типов, принимая во внимание особенности архитектуры Python IDE.

5. Разработать инфраструктуру тестирования и провести сравнения с аналогами.

# 2. Обзор предметной области

Данная глава посвящена обзору предметной области, понимание которой необходимо для дальнейшей реализации собственной подсистемы вывода типов. Так, в данной главе рассматривается архитектура Python IDE, типовая система языка Python, приводится описание принципа работы существующих решений с указанием их достоинств и недостатков, а также рассматриваются алгоритмы вывода типов для языка Python.

## 2.1. Python IDE

SRC IDE — интегрированная среда разработки, поддерживающая язык программирования Java. В данный момент также ведется разработка и добавление поддержки языка Python в SRC IDE. Для дальнейшей реализации вывода типов необходимо понимать общую архитектуру IDE. При этом, поскольку в данной работе рассматривается в первую очередь Python IDE, будет рассмотрена архитектура для нее.

### 2.1.1. Кодовая модель

Как было сказано ранее, кодовая модель — наиболее важный компонент в IDE, отвечающий за представление исходного кода, а также выполняющий анализ и построение семантики языка. Кодовая модель предоставляет API для следующих подсистем IDE: подсистема автодополнения, поиска использований (find usages), навигация, модификация исходного кода (refactorings) и многое другое. Кодовая модель состоит из следующих ключевых компонент:

- Проектная модель (Project Model) — отвечает за высокоуровневое представление проекта. Содержит информацию об исходных файлах проекта, их зависимостях, а также отображает структуру с точки зрения расположения файлов в файловой системе. В случае языка Python также отвечает за конфигурацию Python-интерпретатора и его зависимостей.

- Подсистема синтаксиса — отвечает за представления кода с точки зрения синтаксиса. Для языка Python в рамках данной работы в этой подсистеме был разработан и реализован лексический и синтаксический анализаторы языка Python, поддерживающие как Python 2 версии 2.4 и выше, так и Python 3 всех доступных на текущий момент версий, включая версию 3.11. Синтаксический анализатор основан на методе рекурсивного спуска и поддерживает в том числе и измененную с версии Python 3.9 PEG грамматику языка [4], что позволяет обрабатывать новые конструкции языка, включая сопоставление с образцом, представленный в PEP 622 [5]. Данная подсистема на выходе возвращает конкретное синтаксическое дерево.

- Подсистема семантики — данная подсистема отвечает за построение некоторой семантической информации на основе синтаксического дерева, а именно: построение модельных элементов и специальных высокоуровневых символов — своеобразные высокоуровневые «обертки» над узлами дерева, например, над переменными, функциями, классами. Данные символы впоследствии используются в других подсистемах, например, для показа структуры открытого файла (outline). Также подсистема семантики отвечает за построение пространств имен и областей видимостей различных элементов кода и выполняет процедуру разрешения имен в коде. В эту же подсистему входит и вывод типов.

- Подсистема индексов — данная подсистема отвечает за упаковку и хранение различной информации с дальнейшей целью наиболее быстрого ее восстановления, например, при переоткрытии проекта. В индексах могут храниться в том числе и синтаксические деревья, модельные элементы, информация о разрешенных именах, а также и выведенные типы для выражений языка Python. В Python IDE для этой цели используется специальная база данных.

### 2.1.2. Языковой сервер (Language Server)

Python IDE имеет разделяемую на разные компоненты архитектуру и реализуется с использованием достаточно популярной и современной технологии языкового сервера (Language Server) в отличие, например, от IntelliJ IDEA IDE, имеющей монолитную архитектуру. Впервые данная концепция была представлена компанией Microsoft и получила название «протокол языкового сервера» (Language Server Protocol) [6], являющийся стандартом коммуникации между сервером и клиентом. Принцип работы языкового сервера состоит в следующем: это отдельно запущенный процесс, предоставляющий различные сервисы языка (Language Services), которые взаимодействуют с сервером путем JSON-RPC протокола. Примером такого сервиса может быть сервис автодополнений кода или, например, тот же сервис вывода типов. Принцип работы языкового сервера представлен на рисунке 1 ниже.

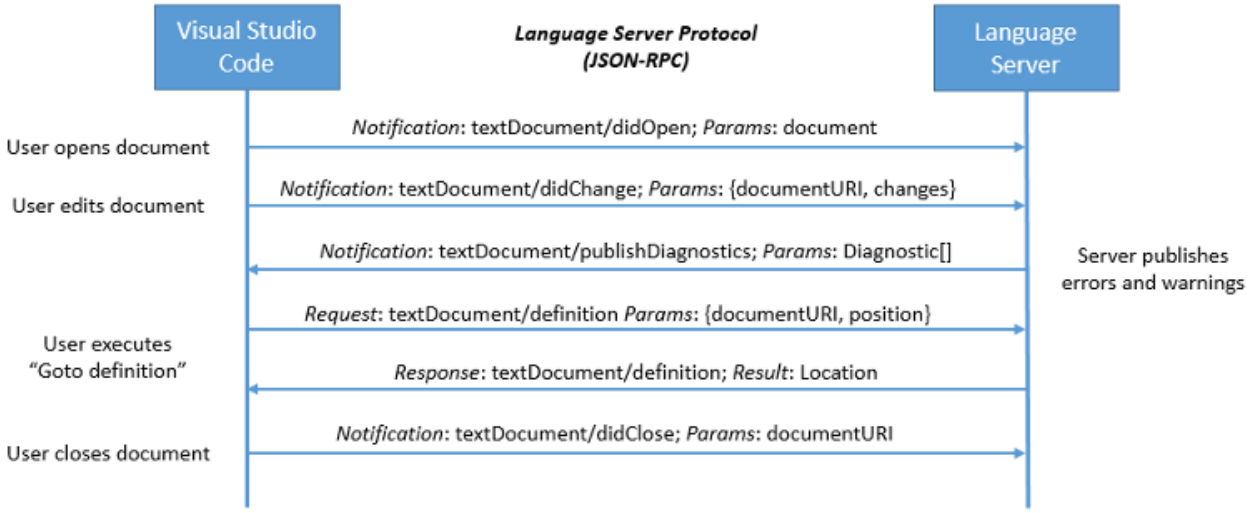


Рис. 1: Пример работы VS Code с использованием Language Server Protocol

Данный подход с использованием языкового сервера в Python IDE позволяет достигать большей гибкости в работе с кодовой базой.

## 2.2. Обзор системы типов языка Python

Прежде чем переходить к обзору существующих решений и алгоритмов вывода типов для языка Python, необходимо рассмотреть устрой-

ство самой системы типов языка.

Стоит учитывать, что в языке Python можно выделить две системы типов: систему типов времени исполнения, то есть систему типов, описывающую действительные типы, которые есть у объектов, в процессе работы/интерпретации программы. Это своего рода «настоящая» система типов языка. Однако в процессе развития языка Python в PEP 484 [7] была представлена альтернативная система типов, основанная на аннотациях (type hints), используемая для так называемой постепенной типизации (gradual typing). Рассмотрим подробнее каждую из них.

### 2.2.1. Система типов времени исполнения

Обратимся к официальной документации языка [8]. Как было сказано ранее, Python — динамический строго типизированный язык, где тип выражения определяется непосредственно при запуске программы в процессе ее работы. Строгость в данном случае означает, что тип значения не меняется некоторым «неожиданным» образом. Например, мы не можем складывать целые числа со строками без дополнительных преобразований, как это можно делать в тех же JavaScript или Perl. Также стоит учитывать, что в языке Python все является объектами, даже константные значения или объявление класса без создания его экземпляра. Отсюда в силу динамической типизации, в языке Python определяется, что непосредственно объекты, а не переменные, имеют тип. Данное правило является немаловажным для реализации статического вывода типов, поскольку благодаря этому можно определять тип переменной с учетом потока управления программы, если выполнять вывод на основе соответствующего графа. Кроме того, в системе типов времени исполнения переменные могут лишь «называть» любой объект. Данная процедура именуется как «связывание имен» (name binding). При этом непосредственно тип объекта определяет доступные над ним операции.

Стандартная иерархия включает следующие типы (для краткости приведены некоторые из них, полный список определен в соответству-

ющем разделе документации [9]):

- None — данный тип имеет единственное значение (singleton), идентифицируемое как отсутствие значения.
- Number — общий тип для численных значений, имеющий следующие подтипы: Integral (int и bool), Real (float) и Complex.
- Sequences types — типы последовательностей, разделяемые на изменяемые и неизменяемые. В неизменяемые входят такие типы, как строки (string), кортежи (tuple), байты (bytes), где коллекция не меняется, даже если меняются элементы в ней. В качестве изменяемых имеются список (list) и массив байтов (bytearray).
- Set types — изменяемый set и неизменяемый frozenset.
- Mappings — единственный тип словаря (dict).
- Callable — к объектам данного типа может быть применен вызов. Это определенные пользователем функции, функции-генераторы, сопрограммы, классы и на самом деле любой объект, имеющий атрибут `__call__`.
- Classes and instances — классы и созданные экземпляры классов.

Система типов времени исполнения обладает следующим видом подтипизации: так называемой «утиной» типизацией (duck typing). «Утиная» типизация основывается на том, что методы/атрибуты объекта важнее типа самого объекта, например, типа класса. То есть в таком примере:

```
def foo(a):
    return a.value
```

нам важно, чтобы в процессе работы программы у передаваемых в функцию объектов был атрибут `value`, а непосредственно сам тип параметра `a` имеет меньшее значение.

### 2.2.2. Система типов постепенной типизации

Данная система типов была впервые представлена в PEP 483 [10] и PEP 484 [7], объясняющих теоретическую часть системы типов и декларирующих разнообразие аннотационных конструкций. Как было сказано ранее, система типов постепенной типизации Python основана на аннотациях типов, используемых в модуле `typing`. Особенностью данной системы типов является то, что она никак не влияет на сам процесс работы программы, интерпретатор попросту игнорирует расставленные аннотации. Главным преимуществом введения постепенной системы типов является то, что она была создана с целью упрощения понимания кода для конечного пользователя, чтобы, в некотором смысле, снизить динамичность языка. Также на основе данной системы типов работают многие статические анализаторы и инструменты проверки корректности типов программы.

Сама система типов постепенной типизации имеет весьма богатый набор аннотационных конструкций, подробнее описанных в документации [11]. Так, поддерживаются описания параметрических коллекций, например, `List[T]`, имеется возможность описания «неоднозначных» или «неточных» типов путем добавления типа `Union` для случаев, когда переменная может принимать значения разного типа. «Неоднозначность» здесь используется в контексте сравнения с системой типов времени исполнения, где, очевидно, значения типа объединения (Union) не может быть в силу однозначного определения типа в момент работы программы. Отсюда можно сделать следующий важный вывод: в данной системе типов единицей типизацией является как раз сама переменная, то есть имя, а не объект. Система типов также имеет такой тип, как `Any` — переменная может принимать значение любого типа.

Также для типов в данной системе применимы понятия контравариантности, ковариантности и инвариантности. Так, например, тип `Tuple` является ковариантным, а значит `Tuple[bool]` является подтипом `Tuple[int]` в силу того, что `bool` является подтипом `int`. Тип `List` же инвариантный и, как следствие, `List[bool]` и `List[int]` не подтипы

друг друга, а тип `Callable` является контравариантным относительно типов своих аргументов и ковариантным относительно типа возвращаемого значения.

Изначально система типов постепенной типизации поддерживала базовый набор аннотаций и учитывала свойство номинальной/номинативной типизации, где определялось, что если A подкласс класса B, то A подтип B. Однако впоследствии в PEP 544 [12] были добавлены протокольные аннотации, вводящие в систему типов структурную подтипизацию, являющейся статической «утиной» типизацией (static duck typing). Благодаря данным протоколом стало возможным описывать типы, имеющие определенный атрибут, например, при использовании аннотации `items: Iterable[int]` считается, что у `items` будет атрибут `__iter__`.

Использование данной аннотационной системы типов приобретает популярность в наше время, однако все равно большинство репозиториев и Python-файлов не имеет широко аннотированного кода, что приведено в соответствующем исследовании [13].

## 2.3. Обзор существующих решений

Прежде чем переходить к реализации вывода типов, необходимо понимать, существуют ли инструменты, обеспечивающие подобную функциональность. Для выполнения данной задачи был осуществлен поиск существующих решений. Найденные решения были рассмотрены на предмет соответствия предметной области в рамках системного обзора литературы. Для определения того, подходит ли найденное решение, был выдвинут следующий критерий выборки:

Найденное решение должно обеспечивать возможность статического вывода типов и их просмотра для выражений языка Python. При этом рассматриваются как решения, интегрированные в различные инструменты разработки, например, в IDE, так и отдельные инструменты или анализаторы исходного кода.

Найденные решения, удовлетворяющие критерию выборки, можно

разделить на две группы:

- Решения, интегрированные в среды разработки или же в компоненты кодовой модели.
- Статические анализаторы Python-кода, проверяющие корректность типов на основе аннотаций, а также реализующие вывод типов.

### 2.3.1. Решения, интегрированные в IDE или компоненты кодовой модели

В рамках данного раздела будет приведено описание и ключевые особенности статического вывода типов, интегрированного в PyCharm IDE, рассмотрена библиотека кодовой модели для языка Python Jedi, а также изучен статический анализатор Pyright, интегрированный в языковой сервер Pylance от Microsoft.

**PyCharm** — IDE для языка Python от компании JetBrains [14]. Имеет собственную кодовую модель, а также семантическое дерево. Вывод типов основан на системе типов постепенной типизации и работает путем рекурсивного обхода семантического дерева. При этом выводятся «ожидаемый» и «действительный» тип, а далее проверяется их согласованность. Вывод «ожидаемого» типа в первую очередь вычисляет тип на основе наличия аннотации или `docstring` — специального формата документирования элементов кода, путем использования собственного инструмента анализа аннотаций. В случае, если тип вывести не удалось, происходит переключение на контекст семантического дерева, и тип выводится с использованием разрешения имен на основе узлов дерева. «Действительный» тип сразу выводится на основе дерева путем наличия у семантических узлов соответствующих методов.

В выводе типов решения PyCharm можно выделить следующие достоинства:

- Система типов PyCharm поддерживает различные коллекции, включает параметризованные типы и объединение типов.

- PyCharm поддерживает типы стандартной библиотеки Python при помощи интерфейсных файлов (stub/typeshed-файлов) — файлов с расширением `.pyi`, где все функции и декларированные переменные уже проаннотированы и имеют тип. При этом сама реализация в данных файлах отсутствует.

- PyCharm поддерживает вывод типов с учетом потока управления программы при помощи построения соответствующего графа потока управления, то есть данный вывод типов является потокочувствительным (flow-sensitive).

- В системе типов также имеется выделенный структурный тип или же тип с так называемой «роу-переменной» (row-variable). Для такого примера:

```
def foo(a):
    a.send_message()
    return a.value
# type(a) = a: {send_message, value}
```

  PyCharm определяет, что переменная `a` имеет структурный тип с атрибутами `send_message` и `value` и впоследствии при помощи специальной инспекции при вызове функции проверяет, есть ли необходимые атрибуты у переданного в качестве аргумента объекта.

Тем не менее PyCharm не способен выводить типы при конкретных применениях, то есть вызовах, функций, параметры которых представляют собой структурный тип. Например, при вызове функции `foo` из примера выше при удовлетворении условию в виде наличия атрибутов результирующий тип всегда будет `Any`. Фактически можно говорить о том, что данное решение способно собирать ограничения в виде наличия атрибутов у параметров функций, однако выполнять их распространение для получения конкретного типа не представляется возможным, что, безусловно, является недостатком, так как не позволяет в полной

мере учитывать динамическую природу языка и свойство «утиной» типизации типовой системы. Помимо этого, в качестве другого недостатка можно выделить отсутствие в системе типов свободных переменных, поэтому для такого примера кода для тождественной функции (функции, возвращающей свой типовой параметр):

```
def foo(a):
    return a

v1 = foo(1)
v2 = foo(``some string'')
v3 = foo(True)
```

PyCharm для всех применений выводит тип `Any`, как и для лямбда-функций.

**Jedi** представляет собой библиотеку [15], содержащую собственную реализацию кодовой модели для языка Python и использующуюся как зависимость в различных языковых серверах, например, в Jedi Language Server или Palantir Language Server. Каждый из данных языковых серверов можно собрать как плагин и использовать в качестве расширения для VS Code или других редакторов кода. Также Jedi может использоваться и в IDE, например, в Spyder IDE. Библиотека Jedi предоставляет достаточно широкую функциональность для анализа Python-кода, имеет собственный синтаксический анализатор, подсистемы автодополнения, рефакторингов, а также возможность вывода типов.

Вывод типов в библиотеке Jedi основан на так называемом «принципе ленивого вывода» и работает поверх собственного синтаксического дерева с использованием разрешения имен. Jedi не выводит типы для всех исходных файлов и их зависимостей, а делает это по запросу и только для тех выражений, которые используются. При этом необходимо понимать, что Jedi основывается на системе типов постепенной типизации, и, например, при наличии аннотации с типом в коде или же

`docstring` у функции, где могут быть явно указаны типы параметров и возвращаемый тип функции, библиотека в таком случае опирается именно на эту информацию.

Также в выводе типов Jedi можно выделить следующие преимущества и недостатки:

- Аналогично выводу типов в решении PyCharm, Jedi поддерживает типы стандартной библиотеки Python при помощи интерфейсных файлов.

- Библиотека поддерживает работу и вывод типов для различных коллекций, например, списков, множеств, кортежей. Однако в силу того, что данные коллекции в общем виде параметрические, то есть имеют типовой параметр, Jedi не поддерживает или не гарантирует корректность типов для некоторых функций данных коллекций, например, `list.append`.

- Jedi также не поддерживает в выводе типов «магические» функции из Python, глобальные переменные с использованием ключевого слова `global`, а также не имеет возможности вывода типов с учетом потока управления за исключением единственной конструкции `isinstance(a, b)`, что, конечно, сужает общее покрытие языка и возможность вывода типов для различных конструкций.

- Помимо этого, Jedi не имеет в типовой системе структурного типа и, соответственно, не предоставляет возможности сбора ограничений в виде наличия атрибутов у параметров функций. Как следствие, вывод типов для данных конструкций, как и обработка случаев динамического добавления атрибутов к объектам, не поддерживается.

**Pyright** — статический анализатор кода и инструмент для проверки и вывода типов от компании Microsoft [16]. Имеет открытый исходный код и входит в состав языкового сервера Pylance от Microsoft, который поставляется в виде плагина к VS Code или другому редактору

кода. Важной особенностью инструмента также является возможность явного вывода типов при отсутствии аннотаций. Система типов Pyright также основана на системе типов постепенной типизации. Вывод типов использует собственное синтаксическое дерево, для которого выполнена процедура построения специальных высокоуровневых символов для классов, функций, переменных, типовых параметров и прочего, а также выполнена процедура разрешения имен и построения областей видимости символов.

Вывод типов в Pyright совмещает концепцию самостоятельного вывода типов выражений, а также учитывает наличие аннотаций и выполняет проверки на корректность типов. В случае, если встречается символ без объявления типа, Pyright пытается определить тип на основе присвоенных ему значений. В системе типов Pyright есть выделенный тип `Uknown` — специальная форма типа `Any`. Тип `Uknown` используется, когда нельзя точно вычислить тип, при этом он может быть впоследствии уточнен при конкретном применении. Рассмотрим следующий пример:

```
def foo(a):
    return a

res = foo(1)  # type(res) = Literal[1]
```

В данном случае возвращаемый тип функции `foo` это `Uknown`, а сама функция `foo` — тождественная функция. Однако при конкретном применении используется техника, называемая выводом типов возврата на месте вызова. При применении функции `foo` происходит уточнение типа путем подстановки типа переданного значения с типом `Uknown`, который переходит в любой другой тип, то есть в данном примере появляется переход `Uknown` $\rightarrow$ `Literal[1]`. То есть тип `Uknown` в некотором смысле является свободной переменной с точки зрения теории типов. Однако стоит учитывать, что подобная техника в данном инструменте не поддерживает лямбда-функции языка Python.

Также для уточнения типа конкретного выражения и устранения

двусмысленностей в выводе типов Pyright может применять различные эвристические подходы или использовать информацию текущего контекста. Данная техника в инструменте называется двунаправленным выводом (bidirectional inference) и основана на использовании «ожидаемого» типа. Рассмотрим следующий пример:

```
var1 = []                    # Type of RHS is ambiguous
var2: list[int] = []         # Type of LHS now makes type of RHS unambiguous
var3 = [4]                   # Type assumed to be list[int]
var4: list[float] = [4]      # Type of RHS - list[float]
```

Как видно из полученного примера, Pyright доуточнил тип выражения для переменных `var2` и `var4` исходя из наличия аннотаций, при этом также производя сопоставления типов.

Из других особенностей можно также выделить поддержку интерфейсных файлов, структурной подтипизации с использованием протоколов, поддержку вывода с учетом потока управления программы. Однако одним из недостатков является отсутствие в системе типов структурных типов или же типов с «роу-переменной», где тип конструируется на основе имеющихся у него атрибутов. Pyright поддерживает структурную типизацию только в случае наличия протокольных аннотаций, представленных в PEP 544 [12]. Поэтому для данного примера будет выведен следующий тип:

```
def foo(a):
    return a.name
# type(a) = Uknown
```

И как следствие, анализатор не сумеет выполнить проверки в случае передачи в функцию объекта, не имеющего атрибута `name`.

Отличительной особенностью данного инструмента является наличие двух режимов вывода: строгий и нестрогий. Строгий вывод типов запрещает, например, переприсваивание в переменную значения другого типа. Нестрогий режим в случае переприсваиваний в переменную значения другого типа конструирует новый тип с использованием типа `Union`.

### 2.3.2. Статические анализаторы кода

Данный вид решений представляет собой отдельные инструменты или же устанавливаемые пакеты, проверяющие корректность кода с учетом типов выражений. Некоторые из них преимущественно основаны на использовании аннотаций типов, а часть имеет собственную типовую систему, на основе которой и производится статический вывод типов и дальнейшая верификация.

**Муру** представляет собой статический инструмент проверки типов для Python-кода [17]. Муру в выводе типов практически полностью опирается на наличие аннотаций. Из данного подхода вытекает один серьезный недостаток: в случае неаннотированного кода для подавляющего большинства языковых конструкций Муру не в состоянии вывести какой-либо конкретный тип и, как следствие, для всех таких конструкций выводится тип `Any`, не представляющий практический пользы, в том числе и для различной функциональности в IDE. Муру может быть интегрирован в различные IDE или редакторы кода, например, путем установки расширения в VS Code.

Муру поддерживает параметрические коллекции, структурную подтипизацию, тип объединения и многое другое, но, как говорилось ранее, данная поддержка осуществляется только при наличии аннотированного кода. Также анализатор придерживается концепции строгой проверки типов: тип выражения не может быть изменен, если это не согласовывается с аннотацией, что показано на примере ниже.

```
a: Union[int, str] = 1
a = ''sds'' #Mypy OK
```

---

```
a = 1
a = ''sda'' #Mypy error
```

Тем не менее в случае неаннотированного кода в Муру была добавлена возможность вывода типов без использования аннотаций, однако она позволяет выводить типы преимущественно при присваиваниях в переменные константных значений или некоторых значений из модуля

встроенных объектов (builtins).

Подводя итог, можно сделать вывод, что данный инструмент больше необходим для проверки типов, нежели их вывода и менее подходит для реализации на основе выведенных типов различной функциональности в IDE, например, квалифицированного разрешения имен или автодополнения кода.

**Pytype** — статический анализатор Python-кода от компании Google, отвечающий за проверку корректности типов в программе [18]. Особенностью данного анализатора является то, что он не полагается только на наличие аннотаций типов в отличие от Муру, а выводит типы самостоятельно, учитывая при этом поток управления программы.

Pytype предоставляет три ключевые функциональности:

- Вывод типов без использований аннотаций с возможностью показа различных ошибок пользователю. Например, Pytype способен показать отсутствие какого-либо атрибута у объекта до запуска программы в случае, если пользователь обратился к подобному атрибуту или же, например, определить некорректный вызов функции при передаче параметра неверного типа.

- Применение и анализ аннотаций типов, используемые пользователем, а также их проверка на корректность в сравнении с собственными выведенными типами.

- Генерация аннотаций типов и объединения с исходным Python кодом.

Pytype основывается на системе типов постепенной типизации, поддерживает вывод типов с учетом потока управления, поддерживает в выводе типов подтипы (subtyping), а также поддерживает типы из интерфейсных файлов для библиотек. В процессе вывода типов анализатор Pytype строит граф потока управления программы, который является главным компонентом для дальнейшего вывода типов. На основе построенного графа в конкретной точке, когда необходимо узнать тип,

Pytype производит обход графа, ищет достижимые узлы и выводит типы для этих узлов.

Важной особенностью при этом является нестрогий вывод типов. Pytype придерживается концепции разрешения всех операций, которые успешны во время выполнения и не противоречат аннотациям. Приведем следующий пример кода, в котором в список добавляются объекты разного типа:

```
from typing import List
 def get_list() → List[str]:
     lst = [''Str value'']
     lst.append(2022)
     return [str(x) for x in lst]

 # mypy: line 4: error: Argument 1 to ''append'' of ''list'' has
 # incompatible type ''int''; expected ''str''
```

Как видно по результатам, статический анализатор Муру не допускает добавления целочисленного объекта в список, который, по мнению анализатора, имеет тип `List[str]` в силу подхода строгого вывода. Однако данный код совершенно корректен с точки зрения работы программы и анализатор Pytype не видит ошибки в данном случае, что является корректным поведением и выводит тип для списка `List[Union[str, int]]`.

Приведем также пример, демонстрирующий вывод типов с учетом потока управления:

```
if some_val:   # 1
  x = 5        # 2
else:
  x = ''a''    # 3
z = x          # 4
```

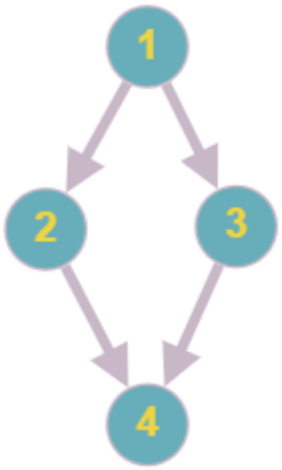


Рис. 2: Граф потока управления для программы из примера

Для данного простого примера и построенного графа потока управления программы для вывода типа переменной `z` Pytype будет рассматривать всевозможные достижимые пути в графе и выводить, какие типы может принимать переменная `x` в данном случае на основе узлов дерева. Конкретно в этом примере тип переменной `z` выведется как `Union[int, str]`. Однако если, Pytype сможет определить, что какое-то условие заведомо ложно, то может вывестись более точный тип.

Некоторые другие примеры сравнения Mypy и Pytype приведены на рисунке 2:

x = *e*

```
def foo(x, y : int):
  i = input()
  if (i == 'Camelot'):
    z = 1
  else:
    z = 'coconut'
#MyPy: ERROR in line 6
#PyType: OK,
#  type(z) = Union[int, str]

  x = 1
  x = 'coconut'
#MyPy: OK, type(x) = Any
#PyType: OK, type(x) = str
```

x.attr = *e*

```
...
a = A(1)
b = a
i = input()
if (i == 'Camelot'):
  a.attr = 'coconut'
else:
  a.attr = 1
#MyPy: OK,
#  type(a.attr) = object
#  type(b.attr) = Any
#PyType: OK,
#  type(a.attr) = type(b.attr)
#  = Union[str, int]
```

x[$e_1$] = $e_2$

```
li = [1]
#MyPy: OK, type(li) = List[int]
#PyType: OK, type(li) = List[int]

li[1] = 'coconut'
#MyPy: ERROR
#PyType: OK,
#  type(li) = List[Union[int,str]]
```

Рис. 3: Сравнение вывода типов Mypy и Pytype из соответствующей статьи [13]

Подводя общий итог анализа инструмента Pytype, можно сделать

вывод, что данный инструмент предоставляет широкий спектр возможностей по статическому выводу типов, а нестрогая модель вывода, в сравнении с тем же Муру, позволяет покрывать больше языковых конструкций и, как следствие, давать более правильные результаты, уменьшая при этом общее количество ложноположительных результатов, что приведено на рисунке сравнения инструментов ниже.

| | false positives | true positives | |
|---|---|---|---|
| | | likely runtime errors | incorrect annotations |
| MyPy | 52 (49%) | 29 (28%) | 24 (23%) |
| PyType | 34 (44%) | 32 (42%) | 11 (14%) |

Рис. 4: Сводка отчетов об ошибках вывода типов Муру и Pytype из соответствующей статьи [13]

## 2.4. Обзор алгоритмов вывода типов для языка Python

В рамках данной подзадачи был осуществлен поиск различных алгоритмов вывода типов. Для поиска алгоритмов и соответствующих статей использовался сервис `Google Scholar`. Найденные статьи были рассмотрены на предмет соответствия предметной области. Также был выдвинут следующий критерий:

Найденный алгоритм/способ должен обеспечивать возможность статического вывода типов для выражений языка Python, основываясь не только на наличиях аннотаций.

Данный критерий обусловлен накладываемыми со стороны Python IDE требованиями и ограничениями, а именно: с точки зрения IDE нельзя полагаться на наличие аннотированного кода, так как пользователю зачастую приходиться работать с еще не проаннотированной кодовой базой. Вывод типов должен уметь работать самостоятельно на основе собственного разработанного семантического дерева.

Таким образом, заявленному критерию не удовлетворяет группа

подходов с конструированием типов на основе аннотаций, как сделано, например, в инструменте Муру, а также группа алгоритмов, основанная на применении машинного обучения. Данный подход с использованием машинного обучения также основан на наличии аннотаций и работает путем обучения моделей для сбора статистики использований и дальнейшего предложения типа с некоторой заданной вероятностью, что, конечно, больше подходит для функциональности автодополнения кода, о чем более подробно рассказывается в соответствующих статьях [19, 20]. Однако данный подход с машинным обучением может быть применен в будущем в Python IDE поверх собственного реализованного вывода типов в ситуациях, когда точный тип неизвестен, и для того же автодополнения можно предлагать наиболее вероятный атрибут.

Рассмотрим решения, удовлетворяющие заявленному критерию:

- Вероятностный вывод типов — данный подход, представленный в статье [21], основывается на выводе типов через использования и по аналогии с алгоритмами, основанными на машинном обучении, выдает вероятностные типы с использованием различных эвристик, что опять же, может быть более полезно для подсистемы автодополнения. В целом, вывод типов через использования является в некотором смысле антипаттерном вывода типов.

- Алгоритм декартового произведения вывода типов (АДП) — впервые представлен в работе О. Агесена [22] и базируется на трех основных идеях: создание типовых переменных, инициализация типовых переменных и сбор ограничений с распространением типов. АДП базируется на работе с мономорфными, а не с полиморфными типами аргументов. Примерами мономорфных типов, то есть конкретных типов, с которыми связано выражение, являются, например, `None`, `int`, `str`. Однако данный алгоритм имеет свои недостатки: не подходит для вывода типа рекурсивных функций, а также не поддерживает полиморфизм по данным, например, список объектов разного типа. В дальнейшем в работе

Бронштейна И.Е [23] было предложено улучшение данного алгоритма, исправляющее описанные проблемы. Однако улучшенный алгоритм, как и первоначальный подход, является нечувствительным к потоку выполнения (flow-insensitive), то есть не позволяет выводить типы с учетом потока управления программы, а также не поддерживает случаи динамического добавления атрибутов к объекту.

- Вывод типов с использованием унификации — данный подход представляет собой процедуру решения уравнений между символьными выражениями. При этом выделяют унификацию разных видов. В случае наличия в унифицируемых термах конструкций высшего порядка, то есть кванторов, говорят об унификации высшего порядка (higher-order unification), а в противном случае — об унификации первого порядка (first-order unification). Задача унификации основана на составлении множества подстановок — отображения из одних типовых переменных в другие. При этом подход с использованием унификации может использоваться и для систем с роу-полиморфизмом — видом полиморфизма, позволяющим писать программы, которые полиморфны на роу-типах: типах записей и полиморфных вариантов (polymorphic variants). В качестве примера языка, имеющего роу-полиморфизм и использующим унификацию, можно привести язык Ocaml.

- Вывод типов с использованием системы верхних/нижних ограничений — оригинальный подход, представленный в статье «Polymorphism, subtyping, and type inference in MLsub» [24], представляет систему типов, сочетающую подтипизацию (subtyping) и параметрический полиморфизм в ML-стиле. Также существует и упрощенная версия данного алгоритма `SimpleSub` [25]. Отличительной особенностью двух данных подходов является расширение системы типов Хиндли-Милнера, а также вывода типов в ней при помощи классического алгоритма унификации, путем добавления в систему подтипизации, что достигается за счет строгого

разделения типов, используемых для описания входов и выходов, а также расширения самого алгоритма унификации и использования алгебры регулярных выражений. Основная идея алгоритма заключается в представлении ограничений на типы не в виде равенств, что получается при помощи классической унификации, а в виде неравенств, позволяющих ограничивать тип сверху и снизу. То есть фактически задача вывода типов для конкретного файла представляет собой решение системы неравенств для получения результирующих ограничений на типы и использует операцию квантификации для определения связанного или свободного вхождения типовой переменной в зависимости от определенного контекста. При этом, поскольку в данной системе есть строгое разделение типов, то имеются следующие инварианты: для описания типов входов, в частности, параметров функции, применима операция пересечения типов, а для описания выходов, то есть возвращаемого типа функции, применима операция объединения. Существует совсем новая работа `MLstruct` [26], расширяющая данную систему и имеющая более сложную и богатую типовую систему. Она снимает ограничения на входные и выходные типы, полноценно добавляет в систему типов поддержку пересечения и объединения, а также поддерживает вывод типов для сопоставления с образцом.

Таким образом, на основе обзора предметной области были приняты следующие ключевые решения:

1. Выбрать в качестве подхода к решению задачи статического вывода типов для языка Python использование подхода с унификацией первого порядка и поиска наибольшего общего унификатора в силу разрешимости данной задачи (нахождение наибольшего общего унификатора или индикация его отсутствия). Данный подход позволяет выводить точные, а не вероятностные типы, в отличие от других рассмотренных моделей. При этом в силу применимости «утиной» типизации к типовой системе языка Python предлагает-

ся учитывать данное требование путем внедрения в собственную типовую систему Python IDE poy-полиморфизма для поддержки poy-типов, чтобы поддерживать вывод типов в конструкциях вида:

```
def foo(a):
    a.send_message()
    return a.value
```

2. Также необходимо учитывать, что унификация позволяет производить строгий вывод типов, то есть, например, изменение типа переменной со строкового типа на целочисленный не унифицируемо. Данные ограничения можно решать путем внедрения вывода типов с учетом потока управления.

3. Системы верхних/нижних ограничений представляют собой весьма интересный как научный, так и практический результат в силу поддержки типов пересечения и объединения. Данные современные работы могут способствовать разработке более совершенных алгоритмов вывода типов, однако в силу новизны для данной предметной области требуют дополнительного изучения и проработки. Тем не менее стоит отметить, что некоторые идеи, в частности с поддержкой типа объединения, могут использоваться в будущем для данной системы.

4. Также необходимо принимать во внимание факт наличия аннотаций в коде, то есть использование системы типов постепенной типизации.

# 3. Архитектура подсистемы вывода типов

Данная глава предназначена для описания принципа работы подсистемы. Также будет рассмотрена спроектированная система типов языка Python, учитывающая свойство «утиной» типизации, случаи динамического добавления атрибутов и параметризацию классов и функций.

Переходя к архитектуре, в первую очередь, необходимо выделить функциональные и нефункциональные требования, а также важно учесть ограничения, накладываемые со стороны Python IDE.

## 3.1. Требования

Подсистема вывода типов должна удовлетворять следующим требованиям:

**Функциональные требования**

- Статический вывод типов должен учитывать одно из главных свойств системы типов времени исполнения — «утиную» типизацию. Необходимо поддерживать вывод в конструкциях с использованием атрибутов, в том числе и для параметров функций, а также поддерживать случаи динамического добавления атрибутов, в частности, к созданным объектам классов.

- Подсистема в процессе вывода должна вычислять параметризацию для классов и созданных экземпляров классов, а также функций, включая функции высшего порядка. Наличие данного требования позволит учитывать, в частности, параметризацию созданных коллекций в исходном коде. Помимо этого, должна быть поддержка вывода для рекурсивных функций, включая взаимную рекурсию.

- Вывод типов должен работать в случаях неаннотированного кода. Данное требование накладывается в виде ограничения со стороны

Python IDE, поскольку для сред разработки, в отличие от статических анализаторов, важно уметь выводить типы самостоятельно, так как исходный код конечного пользователя может быть не аннотирован.

- Необходимо также поддерживать инкрементальный вывод типов. В случае изменений в исходном коде подсистема должна перевыводить типы не во всем проекте, а только в изменившихся файлах и в тех файлах, которые от них зависят.

**Нефункциональные требования**

- Подсистема должна быть разработана на языке программирования Java в силу использования данного языка для SRC IDE.
- Подсистема разрабатывается в среде разработки IntelliJ IDEA.

## 3.2. Система типов

В данном разделе будет приведено описание типов, их структуры и ключевых свойств.

Учитывая требования, сформулированные в разделе 3.1, а также рассмотренные в рамках обзора предметной области системы типов языка Python было принято решение основываться на системе типов времени исполнения, то есть на действительных типах, имеющихся у объектов в процессе работы программы. При этом собственная система типов учитывает стандартную иерархию типов языка Python и коррелирует с ней.

В результате проектирования системы типов был выделен общий интерфейс `IPythonType`, являющийся базовым для любого типа в системе и предоставляющий общий набор методов для работы с типом как для внутренних операций, включая унификацию, так и для внешних клиентов, в частности, подсистемы автодополнения.

В листинге 1 представлен данный интерфейс с соответствующим набором методов.

```
1  public interface IPythonType {
2      List<IPythonType> getParameterTypes();
3
4      List<Pair<IPythonType, IPythonType>>
           congruentTerm(IPythonType term);
5
6      boolean occurs(IPythonType term);
7
8      IPythonType copy();
9
10     IPythonType copy(Map<IPythonType, IPythonType> copies);
11
12     void addMember(String name, IPythonType memberType);
13
14     Map<String, IPythonType> getMembers();
15
16     IPythonType getMemberType(String name);
17
18     default Map<String, List<IPythonDeclaration>>
           getTypeMembersInfo() {
19     ...
20     }
21 }
```

**Листинг 1: Пример общего интерфейса IPythonType для описания любого типа в типовой системе**

Рассмотрим подробнее методы API, имеющиеся у любого типа:

- **getParameterTypes** — данный метод позволяет получить параметризацию типа, представленную в виде списка типов. Метод может вернуть как пустой список в случае, если тип не имеет параметризации, так и список типовых переменных, например, при получении параметризации для типа класса (прим. `A[T0]`). При

этом стоит учитывать, что в качестве типовых переменных могут быть конкретные типы, в частности, для созданных экземпляров классов с выполненной подстановкой (прим. `A[int]`).

- **congruentTerm** — один из основных фундаментальных методов, использующихся для операции унификации. Данный метод возвращает список пар типовых переменных для соответствующих типов. В конкретной реализации данного метода в первую очередь проверяется, являются ли два типа «конгруэнтными»: совпадают ли их декларации, полученные при помощи операции разрешения имен в коде, а также размерности их параметризаций. В случае невыполнения данного условия метод возвращает **null**, означающий, что два типа не унифицируемы друг с другом. Например, при унификации двух функций при выполнении условия конгруэнтности данный метод вернет список пар параметров функций, которые в дальнейшем будут попарно унифицироваться.

- **occurs** — проверка вхождения одного типа в другой. Данная операция необходима для «отлавливания» случая «зацикливания» подстановки. При этом существует два соглашения: считать это ошибкой или нет. В случае данной типовой системы в качестве инварианта и нежелания порождать бесконечные подстановки было принято решение, что если `occurs check` сработал, то унификация не сработала, поскольку в противном случае программа может зависнуть.

- **copy** — данный метод предназначен для копирования типа и используется в процессе вывода при унификации. Ключевым аспектом здесь является то, что благодаря ему становится возможным полиморфный вывод, в частности, при вызовах функций или импортировании каких-либо объектов из другого Python-файла. Полиморфность, то есть создание копии для типа, особенно важна в данном случае, чтобы не «мутировать», то есть не изменять, тип исходной декларации.

- **addMember** — данный метод позволяет добавлять к типу типы его атрибутов и используется, например, при создании типа класса для добавления типов его полей или функций. При этом при помощи данного метода также учитываются случаи динамического добавления атрибутов, в частности, добавления атрибута к созданному экземпляру класса, которого не было в самой декларации класса.

- **getMembers, getMemberType** — данный набор методов позволяет получить или словарь всех типов атрибутов, или тип конкретного атрибута по имени.

- **getTypeMembersInfo** — данный метод является примером использования для внешних клиентов, а именно подсистемы автодополнения и позволяет получить словарь имеющихся атрибутов с именем и декларациями.

По итогу в собственной системе типов были выделены следующие типы, реализующие интерфейс `IPythonType`, представленные на диаграмме классов ниже.

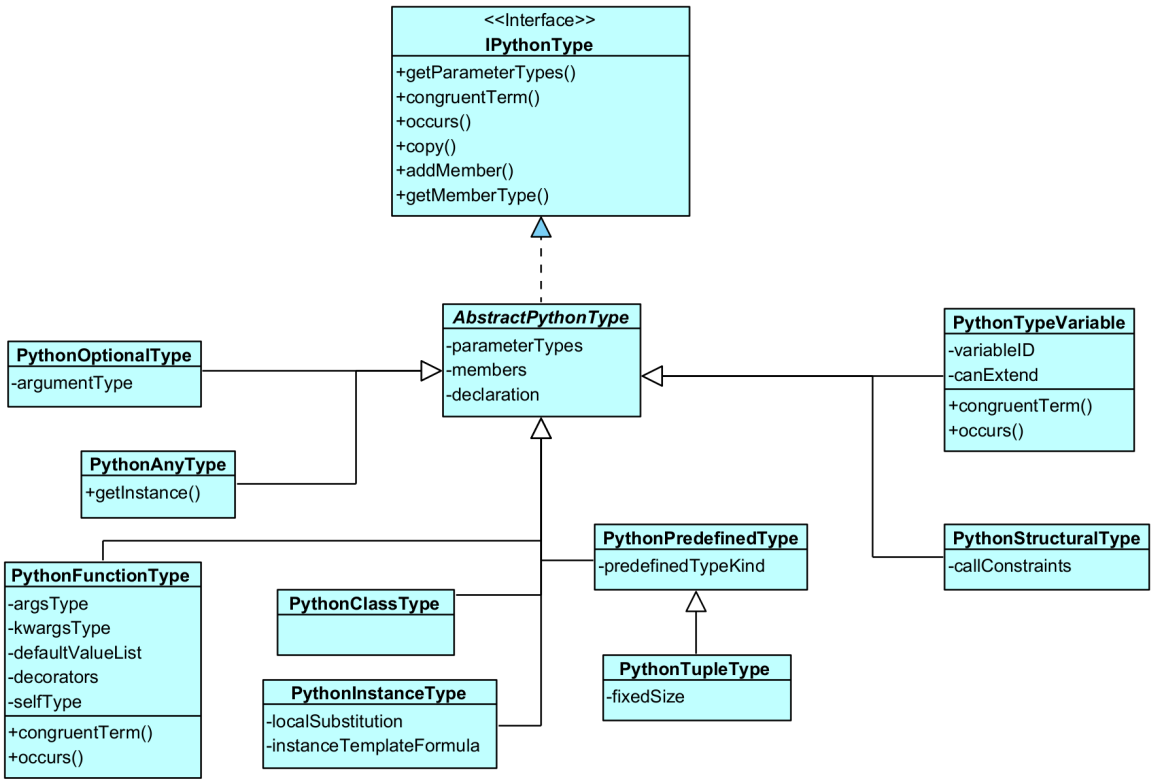


Рис. 5: Архитектура системы типов в виде диаграммы классов

Рассмотрим некоторые особенности и свойства каждого из типов:

- **AbstractPythonType** — является абстрактным классом, реализующим интерфейс `IPythonType`, содержит базовую реализацию методов `occurs`, `congruentTerm` для унификации, хранит список типов параметризации и словарь типов атрибутов.

- **PythonAnyType** — данный тип представляет собой «нижний» (bottom) тип, означающий, что он является подтипом любого типа. `PythonAnyType` не может иметь типов атрибутов и фактически означает, что мы «ничего не знаем про объект данного типа и какие атрибуты у него есть». Создание такого типа происходит в случае ошибки унификации, при этом он реализует паттерн «Одиночка».

- **PythonFunctionType** — представляет собой функциональный тип, предназначенный для соответствия с объектами, которые

могут быть вызваны в процессе работы программы. В частности, функциональный тип создается для функций, объявленных внутри класса, функций высшего порядка в силу их наличия в языке Python, а также для конструкций лямбда-выражений. `PythonFunctionType` помимо типов параметризации расширяется дополнительной типовой информацией: он может содержать типы для специальных конструкций вида `*args` и `**kwargs`, представляющими собой кортеж аргументов переменной длины и словарь именованных аргументов соответственно. Также тип функции хранит необходимую информацию о позициональных и именованных аргументах, поддерживает типы для аргументов, имеющих значение по умолчанию, учитывает наличие декораторов — специальных конструкций, которые могут влиять на поведение функции. При этом методы `occurs`, `congruentTerm` и `copy` реализуются в соответствии с расширенной типовой информацией.

- **PythonClassType** — данный тип предназначен для описания типа класса и помимо параметризации класса хранит информацию о типах статических полей, вложенных классов, функций с учетом первого параметра `self`, так как в случае статического вызова функции первым параметром необходимо передавать экземпляр класса.

- **PythonInstanceType** — предназначен для представления типов созданных экземпляров классов. В качестве типов атрибутов могут содержаться типы как статических полей, так и полей экземпляра класса, а также типы функции. Но при этом стоит отметить, что в случае с функциями, соответствующая для них типовая формула не содержит типа первого аргумента, так как при таком вызове первый аргумент игнорируется.

- **PythonOptionalType** — данный тип позволяет учитывать вывод для объектов с опциональным значением, то есть объектов, которые могут иметь значение конкретного типа или же встроенного

типа `None`, показывающим отсутствие значения. Фактически данный тип можно представить как `Union[`$\alpha$`, None]`. Как показывает практика, в исходном коде поля классов зачастую имеют одну из деклараций в виде присваивания значения типа `None`. Именно поэтому введение данного типа в систему обусловлено требованием по повышению общего качества вывода, а также в некотором смысле строгостью операции унификации, которая в общем случае не унифицирует конкретный тип с `None`, из-за чего происходит переход типа в `Any`.

- **PythonPredefinedType** — предназначен для описания «встроенных» типов, в частности: примитивных типов, таких как типы целочисленных значений, строк, отдельно выделенный тип `None`, так и типов коллекций с соответствующим им набором атрибутов (списки, кортежи, словари, множества и др.).

- **PythonTupleType** — является наследником, то есть в данном случае подтипом, типа `PythonPredefinedType`. Данный тип был выделен как отдельный тип в системе типов для более удобной работы в процессе вывода, в частности, для поддержки вывода с оператором `*`, который позволяет выполнять «распаковку» объектов. При этом `PythonTupleType` обладает следующим типовым свойством: он может быть гомогенным, то есть кортежом одного типа, имеющим переменную длину, или же фиксированной длины с параметризацией в виде разных типов, например, `Tuple[int, float, str]`.

- **PythonTypeVariable** — типовая переменная является одним из ключевых типов в типовой системе, на основе которого становится возможным при помощи операции унификации как вычисление другого типа, так и параметризации типа. Данный тип обладает свойством «перехода» в другой тип в случае «успешной» проверки `occurs`. Типовая переменная может быть как свободной, так и связанной в зависимости от контекста вывода. Связывание типо-

вых переменных может происходить при помощи такой операции теории типов, как квантификация, и позволяет, в частности, поддерживать в различных случаях полиморфный или мономорфный вывод. Стоит отметить, что каждая типовая переменная в данной системе типов имеет свой уникальный идентификатор, не поддерживает операцию `congruentTerm`, а `occurs` проверяет равенство двух идентификаторов.

- **PythonStructuralType** — структурный тип также является одним из важнейших типов в системе, благодаря которому становится возможным удовлетворение одного из главных требований, предъявленных к подсистеме — поддержки свойства «утиной» типизации и вывода типов с учетом использования атрибутов. Данный тип фактически представляет собой тип записи, у которой может быть или не быть определенного набора атрибутов. Введение структурного типа восходит к теории роу-полиморфизма и, таким образом, позволяет представлять тип в виде известной части с набором атрибутов и так называемой переменной расширения — «хвоста» переменной длины, который может расширять тип в процессе унификации. В частности, примером такого типа является параметр функции, на который в процессе вывода были наложены определенные ограничения в виде использования атрибутов.

Таким образом, спроектированная система типов во многом базируется на системе типов Хиндли-Милнера с использованием алгоритма унификации и ее расширением в виде применения теории роу-полиморфизма для поддержки структурных типов. Благодаря данному подходу, стало возможным выводить типы для большинства языковых конструкций Python и, в частности, поддерживать параметрический полиморфизм.

Поскольку задача вывода типов для языка Python неразрешима в общем виде, и данная система типов в том числе имеет свой недостаток в силу определенных ограничений подхода с использованием унифика-

ции: в такой системе типов полноценно невыразим тип объединения (за исключением специально выделенного типа `Optional`, о котором сказано выше), а введение данного типа приводит к неразрешимости унификации и поиска наибольшего общего унификатора, что в некотором смысле «ломает» используемый алгоритм. Из-за данного ограничения на текущий момент не представляется возможным выводить точный тип для полиморфизма по данным, например, в случае списка разнородных объектов. Тип `List[Any]` в таком случае также в некотором смысле корректен, однако в будущем планируется отдельная проработка данного вопроса и поиска возможных решений для поддержки типа объединения, возможно, в том числе путем использования идей из систем верхних/нижних ограничений `MLsub` [24] и `MLstruct` [26] для расширения системы до системы неравенств, что является уже отдельной задачей, выходящей за рамки данной работы.

## 3.3. Принцип работы подсистемы вывода типов

Архитектура всей подсистемы представлена на рисунке 6 в виде диаграммы компонентов.

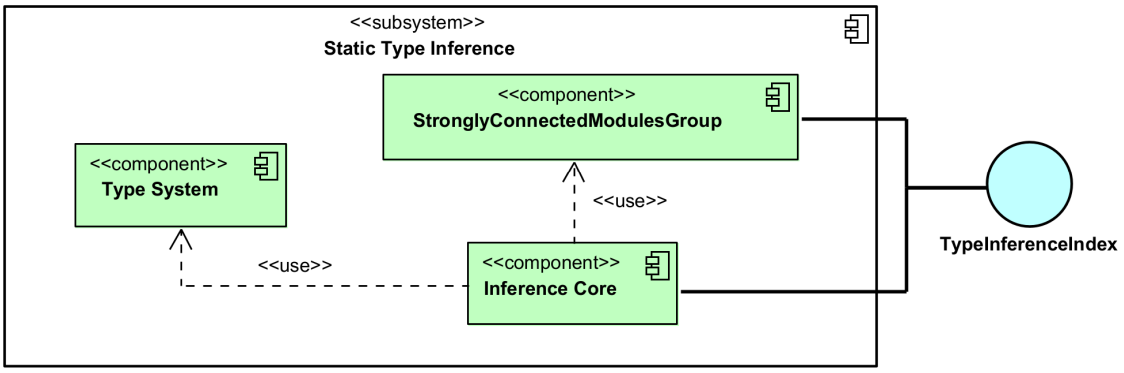


Рис. 6: Архитектура подсистемы вывода типов в виде диаграммы компонентов

Точкой входа в подсистему является специальная сущность `TypeInferenceIndex` — реализованный индекс для получения типов в кодовой модели Python IDE, входящий в подсистему индексов, описан-

ной в разделе 2.1.1. Представляет собой индекс «отображения», то есть ключ-значение, и имеет следующее API, приведенное в листинге 2:

```
1 public interface IPythonTypeInferenceIndex {
2     IPythonType infer(IPythonExpression node);
3
4     IPythonType infer(IPythonDeclaration declaration);
5 }
```

**Листинг 2: Интерфейс IPythonTypeInferenceIndex**

Таким образом, благодаря данным методам API внешние клиенты могут получать выведенный тип, используя либо узел семантического дерева, представляющий какое-либо «выражение», либо при помощи декларации, полученной в результате процедуры разрешения имен в коде.

Рассмотрим подробнее каждую из компонент, а также ее функциональные ответственности:

- **StronglyConnectedModulesGroup** — данная компонента отвечает за создание специальных «групп вывода», которые используются в дальнейшем в процессе вывода типов. Создание таких групп происходит на основе построенного в рамках данной компоненты графа зависимостей модулей, представляющим собой граф компонент сильной связности (бикомпонент) для Python-файлов.

- **Inference Core** — ключевая компонента в подсистеме вывода типов, отвечающая за вывод типов для всех построенных «групп вывода». В ее ответственность входит создание формул и типовых переменных, создание специального списка задач для вывода типов, а также производится сам процесс вывода типов с использованием унификации и теории роу-полиморфизма. В качестве зависимости данная компонента имеет компоненту `Type System`, а также использует результат работы компоненты `StronglyConnectedModulesGroup`.

- **Type System** — компонента, представляющая собой систему типов, описанную в разделе 3.2. `InferenceCore` в процессе работы создает необходимые типовые объекты.

Основной сценарий использования подсистемы вывода типов представлен на диаграмме последовательностей UML (рис. 7) и включает в себя следующие шаги:

1. Внешний сервис, например, сервис автодополнения запрашивает у индекса тип выражения или декларации. В случае наличия уже выведенного типа для данного выражения и его хранения в индексе происходит возвращение выведенного типа.

2. Если тип еще не выведен, а это возможно, если изменилось состояние проекта, например, мы добавили новый набор файлов. В таком случае при обновлении проекта индекс вывода типов инициирует построение графа зависимостей модулей для данного набора файлов, который строится на основе индекса импортов. Также для каждой построенной компоненты сильной связности модулей создается своя «группа вывода», передаваемая дальше в компоненту `Inference Core`.

3. Для каждой «группы вывода» для деклараций создаются свои типовые формулы, производится итеративный процесс с использованием алгоритма унификации, на каждой итерации вычисляется и применяется минимальное множество подстановок, при этом вывод происходит до условия остановки — неподвижной точки. После остановки происходит замыкание формул и возвращение выведенных типов в индекс.

4. После сохранения типов в индекс, сервису возвращается тип запрашиваемого выражения.

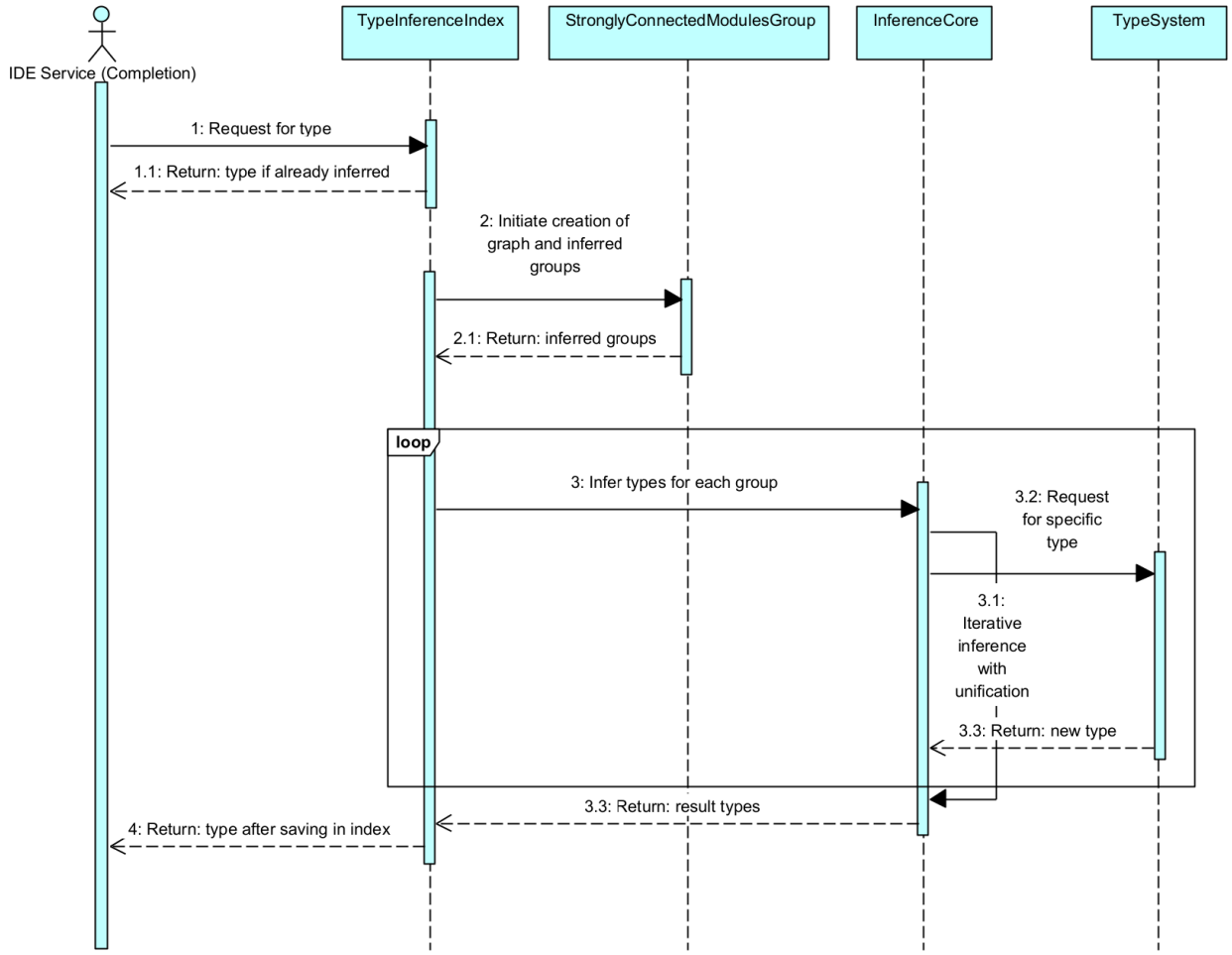


Рис. 7: Основной сценарий использования подсистемы вывода типов в виде диаграммы последовательностей

Таким образом, спроектированная система типов, а также архитектура всей подсистемы позволила учесть сформулированные функциональные требования.

# 4. Особенности реализации подсистемы вывода типов

В данной главе более подробно рассматриваются особенности реализации подсистемы вывода типов, в частности: построение графа зависимостей модулей и создание «групп вывода», процесс построения списка задач, разбирается итеративный процесс вывода типов и приводится сам алгоритм, основанный на унификации и теории роу-полиморфизма с его последующим обоснованием. Помимо этого, в данной главе также затрагиваются выявленные в процессе реализации проблемы и предлагаются пути их решения.

## 4.1. Построение графа зависимостей модулей и создание групп вывода

Как было сказано ранее в описании основного сценария использования системы, `TypeInferenceIndex` инициирует процесс вывода типов, начинающийся с построения графа зависимостей модулей, полученных от проекта.

Построение графа зависимостей модулей основано на индексе импортов, уже реализованном в Python IDE. Индекс импортов представляет собой индекс отображения, сопоставляющий каждому Python-модулю список модулей, от которых он зависит, что выполняется при помощи отдельной процедуры разрешения импортов в коде. Построение графа представляет собой обработку очереди модулей, изначально инициализированной всеми доступными модулями, и выполняется до опустошения очереди. Фактически, обработка каждого модуля выглядит следующим образом: каждый рассматриваемый модуль добавляется в качестве вершины графа, далее рассматриваются все модули, которые он импортирует. Для них также добавляются вершины графа и «проводятся» ребра из зависимого модуля в его зависимости. При опустошении очереди при помощи соответствующего алгоритма производится выделение компонент сильной связности. Ключевым моментом

здесь является именно выделение компонент сильной связности в силу того, что для каждой такой компоненты для ее модулей необходимо производить совместный итеративный вывод. Это обусловлено тем, что в процессе вывода в рамках бикомпоненты типы из одного модуля могут влиять на типы другого, то есть фактически может происходить ситуация доуточнения типами друг друга. Пример построенного графа приведен на рисунке 8.

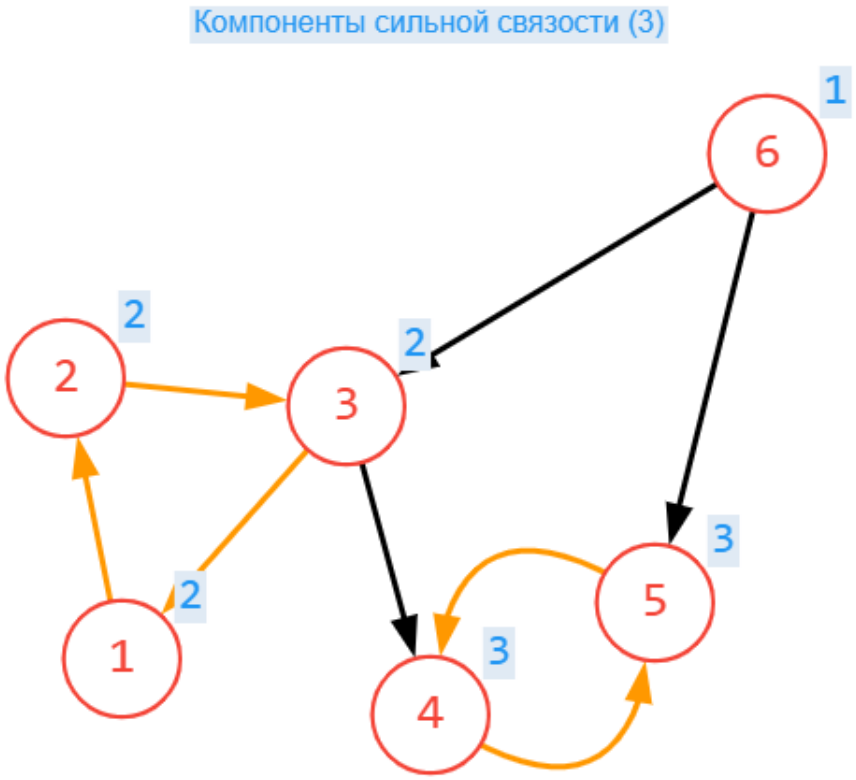


Рис. 8: Пример построенного графа зависимостей модулей с выделенными компонентами сильной связности

При этом подсистема остается устойчивой к случаям циклических импортов, как показано на компоненте номер 3. Хоть фактически в процессе работы программы в данном случае может произойти ошибка, подсистема вывода типов сможет вывести типы при помощи совместного итерационного процесса.

Также стоит отметить удовлетворение требованию инкрементальности: в случае, если изменился определенный файл, граф будет строиться, и, как следствие, будут выводиться типы для него и его зависимо-

стей, а не для всего проекта.

Результатом данного этапа является создание для каждой компоненты сильной связности специальной «группы вывода» — минимальной единицей, в дальнейшем используемой для вывода. Она включает в себя список модулей, свои зависимости в виде других групп вывода, хранит отображение из деклараций в тип, а также отображение из модуля в типовую формулу для него, и помимо прочего, имеет состояние — выведена или нет.

## 4.2. Вывод типов в рамках группы вывода

Результатом работы компоненты `StronglyConnectedModulesGroup` является список созданных «групп вывода» на основе графа зависимостей модулей, который далее передается в компоненту `Inference Core`, где в рамках каждой такой группы происходит вывод типов.

При этом для каждой группы справедливы следующие правила:

- Если группа имеет состояние «выведено», то вывод типов не запускается, поскольку внешние сервисы и клиенты уже могут запрашивать типы деклараций из этой группы.
- В случае, когда группа не выведена, в первую очередь необходимо произвести вывод во всех группах, от которых зависит текущая рассматриваемая группа, поскольку для вывода в текущей группе необходимо знать конечные типы из зависимостей.
- После вывода типов в зависимостях запускается итеративный алгоритм с определенным набором правил для текущей группы вывода, по окончанию которого состояние группы переходит в «выведено».

В данном разделе рассматриваются основные этапы и особенности в рамках вывода конкретной «группы вывода», поскольку это применимо к любой подобной группе.

### 4.2.1. Создание первоначальных формул для деклараций

Первоначальным этапом является этап создания формул для деклараций. Данный процесс выполняется при помощи специальной сущности `FormulasCreatorVisitor` для каждого Python-модуля, входящего в «группу вывода» и основан на получении и обходе в глубину семантического дерева для соответствующего модуля с применением процедуры разрешения имен для получения декларации.

В процессе обхода дерева создаются формулы для следующих деклараций:

- Типы параметров функций — для параметров функций, включая лямбда-выражения языка, создаются соответствующие формулы. При этом параметры функций можно разделить на три группы: именованный параметр, кортеж переменной длины вида `*args` и словарь именованных параметров вида `**kwargs`. Для именованного параметра первоначальным типом является свободная переменная, для кортежа переменной длины — `Tuple[`$\alpha$`]`, а для словаря — `Dict[str,` $\beta$`]`, поскольку тип ключа является фиксированным и представляет собой строковое значение. При этом для всех видов параметров функции соответствующие свободные переменные имеют свойство расширения: в процессе вывода они могут переходить в структурный тип в случае использования атрибутов или динамических вызовов.

- Тип функции — минимальная формула для функции описывается типами своих параметров и типом возвращаемого значения. Данный набор типов является параметризацией функции. Однако при создании формулы для функции также дополнительно учитывается наличие аннотаций вида `staticmethod`, `classmethod`, `property`, так как они влияют на само поведение функции в процессе работы программы. В частности, можно получить результат при обращении к функции, имеющей аннотацию `property`, без ее непосредственного вызова. Помимо этого, формула для функции знает о типах значений по умолчанию, если таковые есть у па-

раметров функции. Возвращаемый тип определяется следующим образом: в случае наличия конструкции `return` без чего-либо, возвращаемый тип определяется как `None`, в противном случае ему присваивается тип свободной переменной, который может уточниться в более конкретный тип.

- Тип идентификатора в присваиваниях — в конструкциях присваиваний, за исключением присваиваний в поля класса, для соответствующих переменных/идентификаторов создаются свободные переменные, при этом не имеющие свойства расширения. Создание данных формул позволяет учитывать типы локальных переменных функций или глобальные присваивания. Отсутствие свойства расширения обусловлено тем, что в конструкциях вида `a = None` более корректно определять, что переменная «a» имеет фиксированный тип `None`, нежели расширять тип до типа `Optional`.

- Тип класса — создание начальной формулы для класса разделяется на создание двух формул: формулы, описывающей конкретно тип класса и содержащей типы статических полей, функций с сохранением первого параметра `self`, а также вложенных классов, и формулы, описывающей «шаблон» экземпляра класса, то есть тип параметра `self`. Формула «шаблона» экземпляра отличается наличием типов полей, а также отсутствием типа первого параметра для всех типов функций класса. На данном этапе для полей класса создаются соответствующие типовые переменные, имеющие свойства расширения: тип поля класса может переходить как в `Optional` в случае наличия декларации с присваиванием значения `None`, так и в структурный тип. При этом создание типовых переменных на данном этапе позволяет впоследствии в процессе вывода вычислять параметризацию класса и экземпляра класса, что является одним из существенных достоинств данной системы.

### 4.2.2. Составление списка задач вывода

Не менее важным является этап составления списка задач. Задача представляет собой определенное «действие вывода» и основывается на синтаксической конструкции языка. Благодаря данному списку задач и становится возможным дальнейший итеративный вывод, который на каждой итерации выполняет все задачи списка.

Построение списка задач, по аналогии с этапом создания формул, основывается на обходе семантического дерева для соответствующего Python-модуля, однако очевидно, что на данном этапе посещаются намного больше узлов дерева, соответствующих синтаксическим конструкциям языка, включая всевозможные виды выражений. Важно отметить и наличие специального стека деклараций, в который добавляются и удаляются элементы в процессе обхода дерева. Данный стек позволяет учитывать текущий контекст, в рамках которого мы находимся при создании новой задачи для конструкции языка. Например, при обработке конструкции вида `return ...` благодаря стеку становится возможным определение функции, возвращаемый тип которой обрабатывается и меняется в данный момент.

Стоит понимать, что «задачи вывода» создаются для подавляющего большинства языковых конструкций. Однако для общего понимания принципа «задачи вывода», а также для краткости будут рассмотрены лишь несколько из них:

- Присваивание — задача вывода для данной конструкции в тривиальном случае выглядит как «вывести тип левой и правой части и унифицировать их». Однако возможны и более сложные случаи: множественные присваивания и присваивания в кортежи. В случае множественного присваивания необходимо вывести тип для каждого идентификатора, в который происходит присваивание, и выполнить унификацию с типом присваиваемого значения. В случае присваивания в кортеж необходимо либо выполнить попарную унификацию, если тип присваиваемого значения также кортеж, либо произвести распаковку (unpacking) для типа при-

свaиваемого значения. В таком случае он должен иметь атрибут `__iter__`.

- Вызов функции — данная задача представляет собой следующее: необходимо вывести тип вызываемого выражения, при этом это может быть как простое имя, так и квалифицированное (например `a.foo(1)`). В случае простого имени достаточно простой процедуры разрешения имен для получения декларации и запрашивания типа. В случае квалифицированного имени необходимо знать «внешний» тип для запроса типа соответствующего атрибута по имени. При этом после получения типа возможны несколько вариантов: полученный тип представляет собой функциональный тип, либо свободную переменную, либо тип класса или экземпляра класса. В случае получения типа класса наблюдается ситуация вызова конструктора, поэтому в таком случае необходимо запрашивать тип атрибута `__new__`. В случае типа экземпляра класса происходит не вызов конструктора, а вызов метода `__call__`, поэтому производится запрос типа данного атрибута, что соответствует поведению программы во время исполнения. Кроме того, если необходимый атрибут отсутствует у класса или экземпляра класса, то благодаря выводу типов становится возможным соответствующий анализ семантики и проверка исходного кода и показывание пользователю ошибки с отсутствием атрибута. Случай получения свободной переменной, имеющей свойство расширения, представляет собой случай динамического вызова функции и приводит к порождению структурного типа и накладыванию определенных динамических ограничений, что будет рассмотрено далее в разделе 4.2.3.

  В корректном случае после получения функционального типа на основе дерева анализируются переданные аргументы: для них выводятся типы, выделяются позиционные и именованные аргументы, а также распознаются использования в качестве аргумента кортежа переменной длины `*args` и именованного слова-

ря `**kwargs`. Результатом является создание специальной сущности `CallConstraint`, агрегирующей информацию об аргументах и имеющей тип возвращаемого значения, представленный в виде свободной переменной. Далее при помощи специального `CallUnificator` производится анализ корректности переданных аргументов в соответствии с формулой функции. Например, становится возможным обнаружение ситуаций неверного количества переданных аргументов или иные случаи. При корректном вызове функции создается формула для вызова и производится унификация с копией формулы исходной функции. Как было сказано ранее, копия исходного типа необходима для того, чтобы конкретные применения, то есть вызовы, не изменяли тип исходной декларации. Таким образом, в случае вызовов функции наблюдается полиморфный вывод типов.

- Конструкция `return` — данная задача представляет собой получение текущего возвращаемого типа функции при помощи обращения к стеку символов, вывод выражения, стоящего после ключевого слова `return` и, наконец, их унификацию. Как можно понять, в случае подхода унификации и наличия нескольких конструкций `return` с возвращением объектов несовместимых типов в силу невыразимости типа `Union` в типовой системе итоговый возвращаемый тип функции становится `Any`.

  Конечно, в определенном случае это является недостатком системы, однако подход с использованием унификации и ее расширения в виде структурных типов позволяет покрывать более широкий класс языковых конструкций, а данный недостаток, как было сказано ранее, предлагается учитывать в будущем и разрешать подобные ситуации.

Отдельно стоит отметить, что для определенных языковых конструкций в «задачах вывода» присутствует логика по обработке использования динамических атрибутов, динамических вызовов и порождения структурного типа, что будет рассмотрено в отдельном разделе

далее.

### 4.2.3. Структурные типы

Для поддержки вывода в случаях с использованием динамических атрибутов, увеличения общего покрытия конструкций языка и удовлетворения одному из сформулированных требований в разделе 3.1, было принято такое ключевое решение, как добавление структурного типа в типовую систему.

Данный тип в первую очередь позволяет «обогащать» тип параметра функции. На него в процессе вывода могут быть наложены определенные ограничения в виде наличия атрибутов. В частности, на примере ниже становится понятным, что без введения структурного типа тип параметра `x` остается свободной переменной. В случае введения соответствующего типа тип параметра представляется как структурный тип, у которого есть атрибуты `name` и `age`. Благодаря этому, также становится возможным анализ и добавление соответствующей кодовой инспекции на предмет наличия у переданного аргумента необходимых атрибутов при вызове функции.

```
def foo(x):
    print(x.name)
    return x.age
# type(x) = x: {name, age}
```

При этом в систему вносится и следующий инвариант: в структурный тип могут переходить только свободные переменные, имеющие свойство расширения. Как было сказано ранее, таким свойством обладают свободные переменные, соответствующие параметрам функции и полям класса.

Рассмотрим ситуации и языковые конструкции, приводящие к созданию структурного типа, а также обработке различного вида динамических конструкций:

- Запрос атрибута — в данном случае возможны две ситуации. В случае, если происходит запрос атрибута от свободной перемен-

ной, имеющей свойство расширения, то происходит создание структурного типа и добавление нового перехода в множестве подстановок вида $\alpha \to$ `{attr_name}`. В случае запроса атрибута от структурного типа и его отсутствия, происходит добавление типа нового атрибута при помощи метода API `addMember`. При этом в обоих случаях добавляемый атрибут имеет тип свободной переменной со свойством расширения, который может перейти или в конкретный тип или также в структурный тип в случае использования соответствующих конструкций.

- Динамический вызов — в данном случае также возможны две ситуации: когда происходит вызов от свободной переменной со свойством расширения и от структурного типа. В случае свободной переменной происходит создание структурного типа и добавление нового правила в множество подстановок. Далее в обоих случаях для структурного типа было принято следующее ключевое решение: при вызове конструируется отложенный `CallConstraint`, который добавляется в соответствующий список для структурного типа. То есть фактически структурный тип описывается динамическими атрибутами и динамическими отложенными ограничениями вызова. Такое решение обусловлено тем, что данный подход позволяет не сужать класс рассматриваемых случаев и выполнять распространение результирующего типа при помощи унификации при конкретном вызове функции.

  Рассмотрим соответствующий пример:

```
def foo(a):
    result = a.send_message("str")
    return result
```

  С одной стороны, одним из решений в данном случае при анализе тела функции было бы накладывание ограничения на атрибут `send_message`, что он является конкретно функциональным типом. Однако это бы привело к существенному сужению класса

возможных случаев, поскольку, учитывая свойство «утиной» типизации языка, в процессе работы программы при вызове функции `foo` под атрибутом `send_message` может находиться объект нефункционального типа, а, например, указателя на класс, что приведет в данном случае к вызову конструктора и созданию экземпляра класса. Именно поэтому решение в виде создания отложенного ограничения вызова и его применения к конкретному типу `send_message` при вызове функции `foo` позволит учитывать подобные ситуации и значительно расширять поддерживаемые случаи.

- Взятие элемента по индексу — данная ситуация сводится к случаю динамического вызова, поскольку на самом деле в процессе работы программы при взятии элемента по индексу происходит запрос атрибута `__getitem__` и его вызов с передачей аргумента. Соответственно, происходит добавление нового ограничения на структурный тип в виде наличия данного атрибута и создание отложенного `CallConstraint`.

- Итерация по элементу — предназначена для вывода типов в конструкции вида `for s in x`. Аналогично сводится к случаю динамического вызова, добавлению к структурному типу атрибутов `__iter__` и `__next__` и соответствующих `CallConstraint`.

- Бинарное выражение — в данной ситуации становится возможным вывод в случаях бинарных выражений и поддержка динамических операторов, таких как логические и арифметические операторы и операторы сравнения. Например, в конструкции вида `x + 1`, где `x` является структурным типом или свободной переменной со свойством расширения, происходит добавление ограничения в виде наличия атрибута `__add__`, что соответствует реальному поведению программы, а также созданию `CallConstraint` с переданным аргументом типа `int`. Таким образом, данные ситуации также сводятся к случаям динамических вызовов функций.

Подобное устройство структурного типа, а также наличие отложенных ограничений вызова позволяет получать точные результаты типов при конкретных применениях. В частности, на примере ниже приведен полученный тип в результате вызова функции, где параметр функции имеет динамические ограничения:

```
class B:
    def bar(self, a):
        return a

class A:
    def __init__(self, a):
        self.a = a
    def foo(self):
        return B()

def foo(x: {foo}):
    return x.foo().bar("ss")

res = foo(A(1)) # type: str
```

В результате вызова функции `foo` при помощи унификации проверяется, что переданный экземпляр класса `A` имеет атрибут `foo`, далее к конкретному типу атрибута `foo`, а также к его возвращаемому результату применяются соответствующие отложенные ограничения вызова, что позволяет получить итоговый тип `str`, что соответствует поведению программы во время исполнения. Стоит отметить, что почти все существующие решения, за исключением Pyright в определенных случаях, не поддерживают вывод в подобных ситуациях и определяют, что результирующий тип — `Any`.

Также отличительной особенностью применения подхода отложенных ограничений вызова к структурным типам является возможность проведения дальнейшего межпроцедурного анализа и анализа корректности вызова до ее исполнения, что приведено на примере ниже:

```
class A:
    def foo(self, x):
        return x

def foo(x: {foo}):
    return x.foo(1, 2)

res = foo(A()) # incorrect call, fail in runtime
               # because A().foo takes one argument
```

Подводя общий итог добавлению структурных типов, стоит отметить, что их совмещение с теорией row-полиморфизма и применение теории унификации позволяет значительно расширить вывод для конструкций языка, а также на их основе реализовывать различные виды анализа семантики программы до ее непосредственного запуска, что является одним из ключевых достоинств системы в сравнении со всеми существующими решениями.

### 4.2.4. Итеративный вывод с вычислением и применением множества подстановок

После построения соответствующего списка задач в рамках группы вывода производится итеративный процесс, где на каждой итерации выполняются следующие шаги:

- Выполняется «действие вывода» для каждой задачи из списка, где применяется унификация, происходит добавление новых правил в множество подстановок, создание структурных типов и добавление отложенных ограничений вызовов.
- По выполнении всех задач из списка проверяется условие остановки алгоритма, фактически представляющее собой неподвижную точку с учетом альфа-эквивалентности типов. При этом для удовлетворения требованием Python IDE, а также в силу неразрешимости задачи вывода типов для языка Python в общем виде,

существует ограничение в виде максимального числа итераций вывода для предотвращения ситуации «ухода в бесконечный вывод», равное 10 итерациям. Данное число обусловлено проведенными тестовыми сценариями на различных библиотеках, включая стандартную библиотеку языка, а также сам системный интерпретатор языка на предмет числа итераций, достаточных для вывода. Результаты показывают, что 7-8 итераций было достаточно для достижения неподвижной точки в данных тестовых сценариях.

- Также в конце каждой итерации происходит применение подстановок ко всем типам, используя накопленное множество подстановок, которое далее очищается.

- После остановки вывода в рамках группы вывода происходит замыкание формул, в частности, уточнение итоговых параметризаций экземпляров классов и типов самих классов.

## 4.3. Алгоритм унификации с row-полиморфизмом

Алгоритм унификации во многом базируется на классическом алгоритме синтаксической унификации, где определяется, что унификация обрабатывает дерево термов (выражений) с метапеременными. При этом в узлах деревьев могут находиться n-местные функциональные символы, а в листьях могут быть так называемые метапеременные. Соответственно, классические правила синтаксической унификации применимы и к данному подходу. В частности, при унификации двух термов в случае, если одному терму уже что-то сопоставляется в накопленной подстановке, то необходимо унифицировать не текущее значение, а значение, которое есть в подстановке, что выполняется при помощи операции `walk`. То есть фактически необходимо унифицировать результат «заглядывания» в подстановку по нужному ключу. При этом в процессе унификации также обрабатываются треугольные подстановки вида $\alpha \to \beta \to \gamma$, которые можно преобразовать в идемпотентные вида $\alpha \to \gamma$ путем замены соответствующего правила в множестве подстановок и

использования операции `walk`.

Далее после применения операции `walk` к двум типам унификация основывается на рассмотрении различных комбинаций данных двух типов и выборе нужного правила унификации. Например, при унификации свободной переменной $\alpha$ с каким-либо конкретным типом необходимо при помощи уже рассмотренной операции `occurs check` проверить вхождение данной типовой переменной в рассматриваемый тип и в случае возможности расширения добавить новое правило в множество подстановок при помощи операции `extend substitution`. Например, при унификации свободной переменной $\alpha$ с экземпляром класса `A[`$\beta$`]` в множество подстановок добавляется правило вида $\alpha \to A[\beta]$, а при унификации с экземпляром класса $A[\alpha]$ расширения не происходит по причине вхождения типовой переменной в рассматриваемый тип. Однако в зависимости от обрабатываемой синтаксической конструкции может произойти расширение вида $\alpha \to Any$. При унификации двух функциональных символов в случае, если их декларации или размерности (арности), то есть количество потомков в поддеревьях, отличаются, то унификация невозможна. В противном случае необходимо попарно унифицировать друг с другом типовые параметры при помощи операции `congruent term` и «протянуть» текущую подстановку через `n` рекурсивных вызовов.

Помимо этого, в алгоритм унификации для данной типовой системы добавляются определенные новые правила. Так, при унификации любого типа с `Any` происходит переход данного типа в `Any`, поскольку тип `Any` является «нижним» типом системы типов и унифицируем с любым рассматриваемым типом. Кроме того, при унификации свободной переменной, имеющей свойство расширения, с фиксированным типом `None` происходит расширение подстановки путем создания типа `Optional`, чтобы выражать опциональность значения, например, для полей класса. При этом при унификации типа `Optional` с каким-либо фиксированным типом, за исключением типа свободной переменной и типа `Any`, необходимо унифицировать аргумент типа `Optional` с рассматриваемым типом. В случае, если рассматриваемый тип является

типом `None`, то множество подстановок расширять не нужно.

Отличительной особенность текущего алгоритма также является применение теории роу-полиморфизма для поддержки структурных типов в системе и соответствующей унификацией с ними. Правила порождения структурных типов, добавление динамических атрибутов, а также обработка динамических вызовов и создание отложенных ограничений вызовов были рассмотрены отдельно в разделе 4.2.3. Унификация с применением теории роу-полиморфизма для структурных типов работает следующим образом:

- В первую очередь, отличием является то, что мы производим унификацию не типовых параметров, а атрибутов, поскольку структурный тип не имеет параметризации. Хотя, в некотором смысле можно говорить, что структурный тип параметризуется типами своих динамических атрибутов и отложенных ограничений вызова.

- При унификации структурного типа с неструктурным типом необходимо сначала проверить, что все имеющиеся атрибуты структурного типа содержатся в другом типе. В противном случае унификация невозможна. Это позволяет, в частности, проверять ситуации, когда в функцию в качестве аргумента передают объект, у которого нет нужного атрибута, имеющегося у параметра функции и представляющего собой структурный тип. В случае успешной проверки необходимо произвести унификацию нужных атрибутов, а также обработать все отложенные ограничения вызова, применяя их к конкретному функциональному типу с помощью уже рассмотренной сущности `CallUnificator` для получения результирующего типа.

- В случае унификации двух структурных типов необходимо выполнить унификацию общих атрибутов, а далее «взаиморасширить» два структурных типа атрибутами и отложенными ограничениями вызова друг друга, что соответствует теории роу-

полиморфизма, где данное расширение возможно благодаря переменной расширения (row-переменной).

В частности, благодаря такому правилу становится возможным «сбор» дополнительных ограничений на использование атрибутов в конструкциях вида:

```
def bar(y : {t}):
    return y.t

def foo(x : {name, t}):
    e = x.name
    return bar(x)
```

Благодаря row-полиморфизму становится возможным «обогащение» структурного типа для параметра функции `x` и добавление требования в виде наличия атрибута `t` помимо атрибута `name`. Обработка подобных ограничений не поддерживается в существующих решениях за исключением PyCharm IDE. Однако стоит отметить, PyCharm способен собрать данные ограничения, но непосредственно получить конкретный тип результата вызова функции не представляется возможным.

## 4.4. Псевдонимы типов

Одной из главных проблем, с которой пришлось столкнуться в процессе реализации подсистемы вывода типов, являлась проблема чрезмерного потребления памяти. Это было обусловлено наличием полиморфного вывода при вызовах функции, то есть созданием полной копии исходной формулы для функции. А поскольку при копировании формулы для функции копируются типы не только параметров и возвращаемого значения, но также типы значений по умолчанию и тип параметра `self` в случае, если мы вызываем метод экземпляра класса, то узким местом системы становилось копирование экземпляров классов, так как тип каждого созданного экземпляра класса содержал полную

иерархию всех типов своих атрибутов.

Для решения проблем потребления памяти и в том числе уменьшения времени вывода была предложена и реализована оптимизация, заключающаяся в подходе с использованием псевдонимов типов (Type Aliases). Основная его идея заключается в том, что каждый класс описывается двумя формулами: формулой для класса и формулой шаблона экземпляра класса (то есть описание типа `self`). При этом созданные экземпляры классов совершенно не хранят типы атрибутов. Они хранят свою параметризацию и собственное локальное множество подстановок из типовых переменных в какой-либо тип, которые расширяются в процессе унификации на основе определенных правил. Помимо этого, тип созданного экземпляра класса хранит ссылку на тип своего шаблона и при запросе типа конкретного атрибута получает тип через шаблон и применяет свою локальную подстановку для получения итогового типа. Таким образом, типы, описывающие экземпляры классов, стали более легковесными, а при их копировании необходимо скопировать только параметризацию.

В качестве результатов оптимизации и применения данного подхода в таблице 1 показано среднее время вывода и максимальное потребление памяти для системного интерпретатора и стандартной библиотеки без подхода псевдонимов типов и с ним.

Таблица 1: Сравнение потребления памяти и времени вывода для системного интерпретатора с применением подхода псевдонимов типов и без

| Вывод типов для системного интерпретатора (∼1800 файлов) | Без псевдонимов типов | С псевдонимами типов |
|---|---|---|
| Максимальное потребление памяти, MB | 5000 - 6000 MB | 800 - 1000 MB |
| Среднее время вывода, мин. | 5 - 6 мин. | 2 - 2.5 мин. |

Подводя общий итог реализованной подсистемы, стоит сказать, что данный подход фактически позволяет производить верификацию исходного кода пользователя, при этом совершенно не полагаясь на наличие аннотаций в коде, что является интересным как практическим, так и теоретическим результатам в сравнении с существующими решениями.

# 5. Апробация решения

Данная глава предназначена для проведения количественной и качественной оценки разработанной подсистемы вывода типов. В рамках главы рассматривается разработанная инфраструктура тестирования, позволяющая выполнять преобразования типов из собственной типовой системы в систему типов постепенной типизации, предлагаются сценарии сравнения с существующими решениями, а также приводятся примеры использования и внедрения получившегося решения в Python IDE.

## 5.1. Тестирование и сравнение с аналогами

Для возможности проведения тестирования подсистемы статического вывода типов было принято решение разработать инфраструктуру тестирования, отвечающую следующим требованиям:

- Инфрастуктура тестирования должна удовлетворять возможности проведения внутренних тестовых сценариев, включая юнит и интеграционные виды тестирований.
- Помимо этого, при помощи данной инфраструктуры должно быть возможным проведение тестирования качества выведенных типов как в сравнении с существующими решениями, так и в сравнении с аннотированной кодовой базой.

### 5.1.1. Инфраструктура тестирования

Для удовлетворения сформулированным требованиям к инфраструктуре было принято следующее решение: взять в качестве основы систему типов постепенной типизации и выполнять преобразования полученных типов в нее, то есть фактически аннотировать код. Данное решение обусловлено тем, что: во-первых, это позволит проводить сравнения с существующими решениями и аннотированной кодовой базой,

поскольку они используют данную систему типов. Кроме того, это открывает доступ в Python IDE к возможности генерации интерфейсных stub файлов, где описываются сигнатуры элементов кода с типами без реализаций. Данная возможность является весьма важной и полезной для среды разработки.

Реализованная инфраструктура основывается на следующих принципах:

- Инфраструктура является параметрической. Это достигается благодаря выделенному параметрическому интерфейсу с соответствующими фабричными методами для создания нужных аннотационных конструкций. В частности, метод для презентации функции в качестве параметров принимает имя функции, типы параметров функции, тип возвращаемого значения, а результат для, например, тождественной функции выглядит как `def foo[T0](x: T0)` $\rightarrow$ `T0`. Параметрическая инфраструктура повышает общую гибкость кодовой базы и позволяет в рамках Python IDE выполнять преобразования из любой типовой системы в единый формат путем реализации конкретного класса.

- Конкретная реализация для системы типов данного решения основана на обходе семантического дерева, а именно его высокоуровневых элементов: функций высших порядков, глобальных присваиваний и классов с внутренней структурой. На основе каждого узла дерева происходит составление соответствующей «презентации» с использованием нужных аннотационных конструкций. При этом, аннотируются только сигнатуры, то есть для тех же функций не приводятся реализации с типами локальных переменных. Таким образом, общим результатом является проаннотированный Python-модуль, содержащий сигнатуры тех элементов кода, которые могут быть импортированы, то есть фактически стала возможной генерация интерфейсного stub файла.

На основе данной инфраструктуры в кодовой модели Python IDE было разработано более двухсот тестовых сценариев с применением

юнит и интеграционного тестирования для оценки качества выводимых типов, покрывающих более 80% конструкций языка Python в соответствии с грамматикой языка [27].

Помимо этого, стало возможным проведение регрессионного тестирования для отслеживания случаев ухудшения выводимых типов путем добавления соответствующего теста, который выводит типы для стандартной библиотеки Python и преобразует их в систему типов постепенной типизации.

### 5.1.2. Выделение тестовых данных и сравнение с аналогами

Качественная и количественная оценка выведенных типов в сравнении с аналогами проводилась в несколько этапов.

В первую очередь, для оценки качества типов были разработаны тестовые сценарии, учитывающие одно из главных свойств языка и требований к данной работе — «утиную типизацию». Для этого были написаны различные тесты с применением конструкций использования динамических атрибутов, вызовов, а также взятия элемента по индексу.

В качестве подобного примера в листинге ниже приводится случай динамического взятия элемента по индексу, где параметр функции `foo` представляет собой структурный тип, на который в процессе вывода были наложены определенные ограничения в виде наличия атрибута `field`, а на результат обращения к атрибуту `field` — в виде наличия атрибута `__getitem__`. По результатам при вызове функции был выведен итоговый тип `int`, что соответствует поведению программы во время исполнения.

```
class A:
    def __init__(self, a):
        self.field = [[a]]

def foo(x):
    return x.field[0][0]
```

```
res = foo(A(1))
```

---

```
class A[T0]:
    self.field: List[List[T0]]
    def __init__(self: A[T0], a: T0) → None
    def __new__(cls: Type[A], a: T0) → A[T0]
def foo[T0](x: {field}) → T0
res: int
```

---

Данный пример является в некотором смысле «модельным» и распространим на более широкие и сложные случаи, тестовые сценарии на которые также были реализованы. Фактически он является иллюстрацией того, что в процессе сравнения с существующими решениями стало понятно, что ни одно из них, за исключением Pyright, не позволяет выводить типы в подобных ситуациях и не поддерживает полноценно свойство «утиной» типизации языка. Статические анализаторы Муру и PyType, как и библиотека Jedi, в принципе не поддерживают в своих типовых системах структурные типы за исключением наличия структурных аннотаций в случае аннотированного кода и, соответственно, не способны выводить типы для динамических конструкций. PyCharm имеет возможность сбора ограничений в виде наличия необходимых атрибутов у параметров функций, но только на один уровень вложенности, и также не позволяет получить какой-либо тип при вызове функции кроме `Any`. Pyright, в свою очередь, в определенных случаях способен выводить типы для данных конструкций, однако не собирает ограничения на параметры функций в виде наличия атрибутов и, соответственно, не позволяет проверять их наличие у переданных аргументов.

Для количественной оценки было принято решение выделить тестовые данные для сравнения и использовать следующую метрику: % типа `Any` относительно всех выведенных типов. Данная метрика аналогична одной из метрик, рассматриваемых в статье [13]. В качестве тестовых данных было принято решение взять библиотеку `Numpy`, по-

скольку она является одной из самых популярных и распространенных библиотек согласно ежегодному исследованию StackOverflow [28], и помимо этого, ее реализация на языке Python не использует аннотаций. Данный факт позволяет провести оценку выведенных типов, не полагаясь на случаи аннотированного кода. Возможностью для проведения сравнения разработанного решения с существующими является генерация интерфейсных stub файлов, поскольку это позволит оценивать типы, используя единый формат представления.

В процессе проведения эксперимента выяснилось, что такие решения как PyCharm и Jedi не имеют соответствующей инфраструктуры и возможности генерации интерфейсных файлов, поэтому были реализованы собственные тестовые сценарии путем модификации исходного кода данных решений. Данные тестовые сценарии включают создание и анализ проекта, получение и обход синтаксического дерева, а также запрос типов. Решение Муру имеет возможность генерации интерфейсных файлов при помощи инструмента `stubgen`, однако поскольку Муру практически полностью полагается на аннотированный код, в случае генерации таких файлов для библиотеки `Numpy` в подавляющем большинстве случаев будет тип `Any`, что не представляет интереса для сравнения. Таким образом, было проведено сравнение с инструментами Pyright, PyCharm и Jedi, результаты которого приведены в таблице 2.

Таблица 2: Сравнение % типа Any относительно выведенных типов с существующими решениями

| Метрика | Статический вывод типов в Python IDE | Pyright | PyCharm | Jedi |
|---|---|---|---|---|
| % типа Any относительно выведенных типов | 9.09 % | 14.01 % | 16.41 % | 22.6 % |

Помимо этого, был проведен и другой эксперимент со следующей метрикой: % разрешенных квалифицированных имен в коде. Данный эксперимент является показательным для оценки общего качества выведенных типов, а также позволяет в некотором смысле отразить то,

насколько типизируется программа и Python-проект в целом.

Таблица 3: Сравнение % разрешенных квалифицированных имен в коде в сравнении с существующими решениями

| Метрика | Статический вывод типов в Python IDE | Pyright | PyCharm | Jedi |
|---|---|---|---|---|
| % разрешенных квалифицированных имен | 61.51 % | 27.22 % | 58.32 % | 52.1 % |

По итогам проведенных экспериментов, можно сделать вывод, что выбранный подход с использованием унификации, а также его расширения в виде использования теории роу-полиморфизма, является оправданным и эффективным и позволяет типизировать большее количество конструкций языка в сравнении с существующими решениями. Вероятно, для дальнейшего уменьшения % `Any` в программе необходима полноценная поддержка типа `Union` в типовой системе, что, как говорилось ранее, является задачей на будущее.

## 5.2. Внедрение

В качестве внедрения подсистемы статического вывода типов в Python IDE была реализована функциональность в виде отображения типа выражения или типа переменной при наведении курсора в открытом документе. Данная функциональность была добавлена на сторону языкового сервера Python IDE в виде соответствующего сервиса, предоставляющего типы и отвечающего необходимым архитектурным требованиям, более подробно описанным в разделе 2.1.2. Также благодаря данному сервису в Python IDE стала возможной квалифицированная навигация для выражений вида `a.b`, поскольку, зная тип переменной `a` и типы его атрибутов, можно получить тип атрибута `b` и соответствующую ему декларацию.

Помимо этого, была произведена интеграция с подсистемой автодополнения, вследствие чего стало возможным отображение пользовате-

лю имеющихся атрибутов при использовании точки после имени переменной. Пример такой интеграции приведен на рисунке 9 ниже.

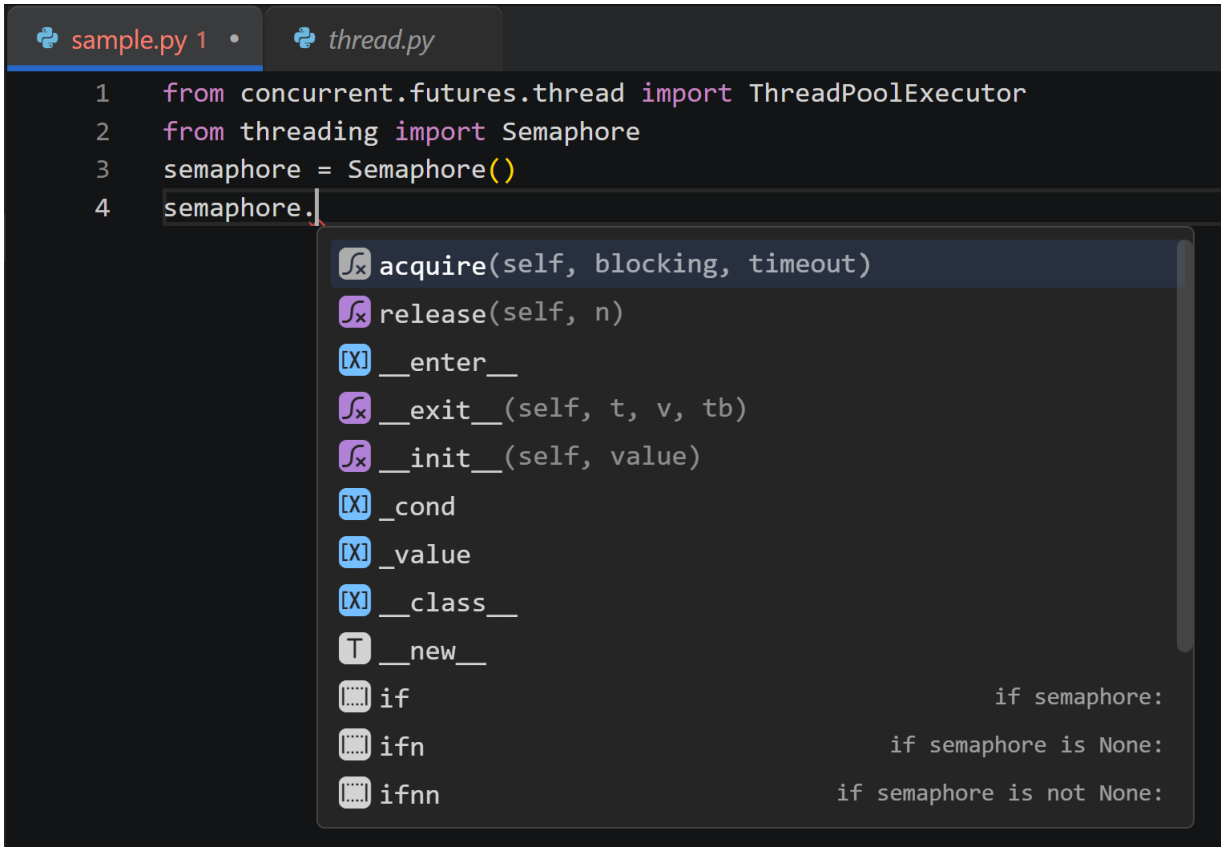


Рис. 9: Пример интеграции вывода типов с сервисом автодополнения

Таким образом, использование подсистемы статического вывода типов, а также различной функциональности, зависящей от данной подсистемы, позволит улучшить пользовательский опыт разработки на языке Python в Python IDE.

# Заключение

В ходе выпускной квалификационной работы были достигнуты следующие результаты.

- Выполнен обзор предметной области:
  - изучена система типов языка Python, включая ее различные подвиды, а также рассмотрены существующие целевые решения: PyCharm, Jedi и Pyright (интегрированы в IDE), Муру и PyType (статические анализаторы);
  - выявлены ключевые недостатки существующих решений, в частности, отсутствие поддержки или неполнота вывода типов для конструкций с использованием атрибутов;
  - рассмотрены различные подходы и алгоритмы вывода типов с применением машинного обучения, теории вероятностей, теории унификации, системы верхних/нижних ограничений, выбран подход на основе теории унификации и системы с роу-полиморфизмом для поддержки свойства «утиной» типизации Python.
- Спроектирована система типов в рамках Python IDE с учетом выявленных требований, предоставляющая структурные типы с поддержкой атрибутов, параметрические и функциональные типы.
- Реализована подсистема статического вывода типов языка Python в рамках Python IDE, учитывающая вывод для типов с атрибутами, включая случаи их динамического добавления и динамические вызовы, а также параметризацию классов и функций.
- Выполнена апробация подсистемы:
  - разработана инфраструктура тестирования созданной подсистемы с представлением типов в едином формате с использованием системы типов постепенной типизации;

– проведены качественные и количественные сравнения с существующими решениями PyCharm, Pyright и Jedi с использованием метрик процента типа `Any`, а также подсчетом разрешенных квалифицированных имен в коде;

– реализована функциональность на стороне языкового сервера в виде отображения типов, а также произведена интеграция с подсистемой автодополнения.

По итогам полученных результатов, виден дальнейший вектор работы над подсистемой статического вывода типов со следующими задачами:

1. Добавить полноценную поддержку типа `Union` в собственную типовую систему и поддержать вывод типов в случаях с возвратом из функции объектов несовместимых типов (не подтипы друг друга).

2. Добавить поддержку номинальной типизации (subtyping), учитывающую конверсию типов, то есть повышающее и понижающее приведение типов.

# Благодарности

Автор выражает отдельную благодарность коллегам, помогавшим при разработке данной подсистемы и работы в целом.

- Тропину Николаю Владимировичу — за всестороннюю помощь в решении архитектурных вопросов, а также проверку написанного автором исходного кода.

- Лукьяновой Ольге Евгеньевне — за написание рецензии на текст выпускной квалификационной работы, а также за то, что, будучи лидером команды, позволила автору взяться за данную нетривиальную работу и активно участвовала во всех соответствующих встречах и процессах принятия решений.

- Степанову Семену Алексеевичу — за помощь в разработке подсистемы и ее интеграции с реализованным промежуточным представлением кода в виде Static Single-Assignment form, что позволило выводить типы с учетом потока управления программы, а также повысить общее качество выводимых типов.

Помимо этого, автор благодарит своего научного руководителя Булычева Дмитрия Юрьевича за ценные советы, связанные с теоретической стороной вывода типов и типовыми системами, за помощь в создании и проверки текста, а также за контроль прогресса работы в целом.

# Список литературы


[1] IntelliJ IDEA IDE from JetBrains. — URL: https://www.jetbrains.com/ru-ru/idea/ (дата обращения: 15 декабря 2022 г.).

[2] Microsoft Visaul Studio. — URL: https://visualstudio.microsoft.com/ru/ (дата обращения: 15 декабря 2022 г.).

[3] Stack Overflow 2022 research on most popular programming languages. — URL: https://survey.stackoverflow.co/2022/#most-popular-technologies-language (дата обращения: 22 декабря 2022 г.).

[4] Ford Bryan. Parsing Expression Grammars: A Recognition-Based Syntactic Foundation // Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. — POPL '04. — New York, NY, USA : Association for Computing Machinery, 2004. — P. 111–122. — URL: https://doi.org/10.1145/964001.964011.

[5] Python PEP 622: Structural Pattern Matching. — URL: https://peps.python.org/pep-0622/ (дата обращения: 16 февраля 2023 г.).

[6] Language Server Protocol from Microsoft. — URL: https://microsoft.github.io/language-server-protocol/ (дата обращения: 16 февраля 2023 г.).

[7] Python PEP 484: Type Hints. — URL: https://peps.python.org/pep-0484/ (дата обращения: 16 февраля 2023 г.).

[8] The Python Language Reference. — URL: https://docs.python.org/3/reference/ (дата обращения: 16 февраля 2023 г.).

[9] The standard Python type hierarchy. — URL: https://docs.python.org/3/reference/datamodel.html#

the-standard-type-hierarchy (дата обращения: 16 февраля 2023 г.).

[10] Python PEP 483: The Theory of Type Hints. — URL: https://peps.python.org/pep-0483/ (дата обращения: 16 февраля 2023 г.).

[11] Python module typing: Support for type hints. — URL: https://docs.python.org/3/library/typing.html (дата обращения: 21 марта 2023 г.).

[12] Python PEP 544: Protocols: Structural subtyping (static duck typing). — URL: https://peps.python.org/pep-0544/ (дата обращения: 21 марта 2023 г.).

[13] Python 3 Types in the Wild: A Tale of Two Type Systems / Ingkarat Rak-amnouykit, Daniel McCrevan, Ana Milanova et al. — DLS 2020. — New York, NY, USA : Association for Computing Machinery, 2020. — P. 57–70. — URL: https://doi.org/10.1145/3426422.3426981.

[14] PyCharm Python IDE from JetBrains. — URL: https://www.jetbrains.com/ru-ru/pycharm/ (дата обращения: 15 декабря 2022 г.).

[15] Jedi: an awesome autocompletion, static analysis and refactoring library for Python. — URL: https://jedi.readthedocs.io/en/latest/index.html (дата обращения: 21 марта 2023 г.).

[16] Pyright: static type checker for Python from Microsoft. — URL: https://github.com/microsoft/pyright (дата обращения: 21 марта 2023 г.).

[17] Mypy: a static type checker for Python. — URL: https://mypy.readthedocs.io/en/stable/index.html (дата обращения: 21 марта 2023 г.).

[18] Pytype: a static type analyzer for Python code from Google. — URL: https://google.github.io/pytype/ (дата обращения: 21 марта 2023 г.).

[19] Cui Siwei, Zhao Gang, Dai Zeyu et al. PYInfer: Deep Learning Semantic Type Inference for Python Variables. — 2021. — 2106.14316.

[20] Type4Py / Amir M. Mir, Evaldas Latoškinas, Sebastian Proksch, Georgios Gousios // Proceedings of the 44th International Conference on Software Engineering. — ACM, 2022. — may. — URL:

[21] Python Probabilistic Type Inference with Natural Language Support / Zhaogui Xu, Xiangyu Zhang, Lin Chen et al. — FSE 2016. — New York, NY, USA : Association for Computing Machinery, 2016. — P. 607–618. — URL: https://doi.org/10.1145/2950290.2950343.

[22] Agesen Ole. The Cartesian Product Algorithm: Simple and Precise Type Inference Of Parametric Polymorphism // Proceedings of the 9th European Conference on Object-Oriented Programming. — ECOOP '95. — Springer-Verlag, 1995. — P. 2–26.

[23] Bronshtein Igor. Type inference for Python programming language // Proceedings of the Institute for System Programming of RAS. — 2013. — 01. — Vol. 24. — P. 161–190.

[24] Dolan Stephen, Mycroft Alan. Polymorphism, subtyping, and type inference in MLsub // ACM SIGPLAN Notices. — 2017. — 05. — Vol. 52. — P. 60–72.

[25] Parreaux Lionel. The Simple Essence of Algebraic Subtyping: Principal Type Inference with Subtyping Made Easy (Functional Pearl) // Proc. ACM Program. Lang. — 2020. — aug. — Vol. 4, no. ICFP. — 28 p. — URL: https://doi.org/10.1145/3409006.

[26] Parreaux Lionel, Chau Chun Yin. MLstruct: Principal Type Inference in a Boolean Algebra of Structural Types // Proc. ACM Program.

Lang. — 2022. — oct. — Vol. 6, no. OOPSLA2. — 30 p. — URL: https://doi.org/10.1145/3563304.

[27] Python full grammar specification. — URL: https://docs.python.org/3/reference/grammar.html (дата обращения: 19 апреля 2023 г.).

[28] StackOverflow Developer Survey 2022: Most popular technologies. — URL: https://survey.stackoverflow.co/2022/#section-most-popular-technologies-other-frameworks-and-librarie (дата обращения: 26 апреля 2023 г.).