UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft
Faculty for Computer Science, Electrical Engineering and Mathematics
Department of Computer Science
Research Group Secure Software Engineering

Master's Thesis Proposal

Submitted to the Secure Software Engineering Research Group
in Partial Fulfilment of the Requirements for the Degree of

Master of Science in Computer Science

TypeEvalPy

—

Common Evaluation Framework with Benchmarks for Type Inference

by
Samkutty Sabu

Thesis Advisor:
Ashwin Prasad Shivarpatna Venkatesh

Thesis SupervisorS:
Prof. Dr. Eric Bodden
Dr. rer. nat. Stefan Dziwok

Paderborn, March 30, 2023

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen worden ist. Alle Ausführungen, die wörtlich oder sinngemäß übernommen worden sind, sind als solche gekennzeichnet.

Ort, Datum	Unterschrift

Abstract. Type inference is a key feature in static analysis that helps developers detect errors early, improve code quality, and enhance maintainability. Python is a dynamically-typed language, which means that types are inferred at runtime, making it challenging for developers to ensure type safety. Several type inference tools have been developed to help overcome this issue. However, the effectiveness and efficiency of these tools depend on various factors, such as the complexity of the code and the nature of the problem being solved. Therefore, it is crucial to have a benchmark framework to evaluate and compare the performance of these tools. Existing research has primarily focused on evaluating type inference tools using real-world benchmarks or custom benchmarks created by the researchers.Although these methods are helpful, they do not consider the significance of micro-benchmarks. Additionally, there is a lack of a standardized result format that can be used to compare and interpret benchmark results across all tools. To overcome these challenges, the Benchmark for Type Evaluation (TypeEvalPy) proposes a common evaluation framework for type inference tools. This framework incorporates both micro-benchmarks and real-world benchmarks, which together can provide a comprehensive and accurate assessment of the tools' performance. Furthermore, it includes a common framework for analyzing tools that will enable easier comparison of results obtained from different tools.

Contents

1 Introduction

Type inference is a crucial part of static analysis, and it plays a vital role in improving the quality and reliability of software. Python is a dynamically typed programming language, which means that the data type of a variable is determined at runtime rather than at compile-time. This dynamic nature of Python poses a challenge for static analysis tools as they cannot determine the data type of variables without executing the code. As a result, type inference in Python is a complex and challenging problem.

There has been significant research in developing type inference tools for Python [1, 2, 3, 4, 5, 6, 7, 8, 9]. These tools aim to improve the efficiency and accuracy of type inference in Python. However, evaluating the performance of these tools is a challenging task due to difficulty in creating a reliable ground truth for evaluation. Unlike statically-typed languages where types are explicitly declared, types in dynamic languages like Python are inferred implicitly, making it difficult to establish ground truth.. Evaluating the performance of type inference tools on real-world programs can further complicate the evaluation process. Such programs are often large and complex, requiring significant time and resources to evaluate accurately.

Micro-benchmarks can be a powerful method to evaluate the performance of type inference tools, especially for complex features in Python. These benchmarks are set of small code snippets that can be used to test specific language features and constructs. These code snippets are designed to be easy to execute and evaluate, and they can be used to quickly identify the strengths and weaknesses of type inference tools. These benchmarks are essential because they can help identify performance issues in specific language constructs that may not be apparent in larger programs. Moreover, micro-benchmarks can be used to evaluate the performance of different type inference tools against specific language constructs, making it easier to identify which tool performs better.

Despite the benefits of micro-benchmarks, most developers and researchers only rely on real-world benchmarks to evaluate the performance of type inference tools[1, 2, 3, 4]. Real-world benchmarks are often considered a more accurate representation of the performance of type inference tools as they evaluate the tools on real-world programs. However, using real-world benchmarks for evaluation can be challenging due to the complexity of the programs and the time and resources required to evaluate the tools.

Another issue that needs to be addressed is the lack of a common result format for type inference tool evaluations. Existing research often reports tool performance using different metrics, making it challenging to compare and interpret the results of different studies. For instance [10] used MRR metrics which is connected to score of prediction according to ranking of types while [4] used precision and recall. Therefore, it is crucial to establish a common result format

that can be used by all researchers to facilitate comparison and interpretation of the benchmark results for all tools.

This research aims to provide a benchmark framework for evaluating type inference tools for Python that incorporates both micro-benchmarks and real-world benchmarks. The primary objective of this research is to develop a comprehensive and standardized benchmark framework that can be used by developers to evaluate the performance of type inference tools. The framework also provides a common result format for tools analysis and results, making it easier to compare and interpret the results of different studies.

To achieve this objective, this research has three main goals:

1. Developing a micro-benchmark: The first goal is to develop a micro-benchmark for evaluating the performance of specific language features and constructs in Python. This will involve identifying a set of commonly used language features and constructs and developing code snippets to test them.

2. Standardizing real-world benchmarks: The second goal is to aggregate existing real-world benchmarks and extend them to cover a wide range of Python code. This will involve identifying a set of real-world Python programs and developing test cases to evaluate the performance of type inference tools on these programs.

3. Developing a common framework: The third goal is to develop a common framework for evaluating type inference tools that incorporate the micro-benchmark and real-world benchmarks. This framework will provide a standardized way of evaluating the performance of type inference tools and will also include a common result format for reporting the results of the evaluations.

The research questions that will be addressed as part of this research are:

1. How do existing type inference tools for Python perform on the developed micro-benchmark?
   (a) Does deep learning-based tools support complex python features?
2. How does existing tools perform on the standardized real-world benchmark?
   (a) Does existing real-world benchmarks capture complex features of Python?

In summary, this paper proposes a benchmark framework for evaluating type inference tools that includes both micro-benchmarks and real-world benchmarks. The framework will provide a common result format for tools analysis and results and evaluate the performance of type inference tools. The proposed benchmark framework will be built on top of Scalpel, a static analysis tool for Python code. Scalpel has been used in various research studies to evaluate the performance of type inference tools. Therefore, using Scalpel as the basis for the proposed benchmark framework will provide a solid foundation for the research.

2

Background

Static analysis is a technique used in software development to detect programming errors and improve software quality. In static analysis, code is analyzed without executing it, which can help detect a wide range of issues such as syntax errors, runtime errors, and logic errors. One of the key components of static analysis is type inference, which involves determining the types of variables and expressions in a program without running the program.

Type inference is particularly important in statically typed languages such as Java and C++, but it is also becoming increasingly popular in dynamically typed languages such as Python. Type inference in Python can help developers catch type-related errors at compile-time instead of at runtime, which can lead to more robust and reliable code.

While many type inference tools are available for Python, their performance can vary significantly depending on the complexity of the code, the types of variables used, and other factors. This variation can make it challenging for developers to evaluate the performance of different type inference tools effectively. It also highlights the importance of benchmarking and evaluating type inference tools to determine their strengths and limitations in different contexts.

To address this challenge, several real-world benchmarks have been developed for evaluating the performance of type inference tools [11, 10, 1, 3, 4]. These benchmarks use large datasets of real-world Python code to evaluate the accuracy and efficiency of different types of inference tools. By providing a standardized way of evaluating the performance of these tools, these benchmarks can help developers choose the most appropriate type of inference tool for their needs and improve the overall quality of their software.

2.1 Type inference tools for Python:

Python is a popular programming language used by developers worldwide. Several types of inference tools are available for Python and these tools vary in their approach and effectiveness, with some focusing on specific types of programs or language features.

Academic Type Inference Tools for Python:

1. PYInfer [1]: PYInfer is a type inference tool that automatically generates type annotations. The tool uses PySonar2 to generate initial annotations from top-star Python GitHub projects. The Python source code from 4,577 top-star GitHub repositories contains 77,089,946 annotations, covering user-defined types, 11 basic types, and 500 common

types. PyInfer collects contextual information about variable usage within its scope, and using a neural network predicts the variable's type.

2. HiTyper [2]: HiTyper is a type inference tool that uses both static analysis and deep learning techniques to predict type of variables. It creates a type dependency graph(TDG) to aid inferring types in Python code. The core idea of this approach is to capture the relationships between variables and encode them into type dependency graphs (TDGs) that capture the type dependencies among variables in each function. These TDGs allow for the easy integration of type inference rules to perform static inference and type rejection rules to verify the correctness of the neural network's predictions. The HiTyper tool iteratively combines static inference and deep learning-based prediction until the TDG is fully inferred, enabling more accurate type inference. By leveraging both static inference and neural networks, HiTyper is able to improve the accuracy and performance of type inference tools.

3. Typilus [3]: Typilus is a type inference tool that uses neural networks to infer types in Python code. Typilus uses a graph neural network to predict types in a program's abstract syntax tree. The tool is trained on a dataset of Python programs with type annotations extracted from the PyPI repository. The dataset contains over 100 million lines of code, and the types in the corpus generally follow a fat-tailed Zipfian distribution, with 32% of types being rare.

4. Typewriter [4]: Typewriter, developed by Facebook, can automatically add type annotations to code that is missing them. It takes in code that has some types already labeled and extracts information from it, including the structure of the program and natural language information like comments and variable names. Then, a neural network learns from this information and the labeled types to predict the missing types in the code. Finally, TypeWriter uses a feedback-directed search to ensure the predicted types are consistent and correct according to a static type checker. The result is code with additional type annotations Typewriter has been evaluated on a multi-million line codebase at Facebook and 1,137 popular open-source projects with 50+ stars.

5. TypeT5 [12]: The TypeT5 model can predict the types of code elements by constructing a usage graph that represents potential user-usee relation between them. The graph is then used to generate additional context information for the transformer model, which is based on the popular CodeT5 architecture. The approach claims to perform better compared to earlier approaches on rare or complex types and produces results with fewer type errors.

Other type inference tools: In addition to the academic type inference tools mentioned previously, several other popular tools are used for static analysis and type inference in Python.

1. Mypy [5]: Mypy is a static type checker for Python that allows developers to add type annotations to their codebase and gradually improve the type coverage of their project. It is capable of inferring the types of most variables based on their initial assignments. However, mypy will not use type inference in dynamically typed functions (those without a function type annotation). It combines the benefits of dynamic (or "duck") typing and static typing and allows for type annotations to be gradually added to a codebase. Mypy supports a variety of syntax for specifying types, including Union types and TypeVars, and also supports plugins to extend its functionality.

2. Pyright [6]: Pyright is a fast and accurate type checker for Python that is developed by Microsoft. While Pyright attempts to infer the type of a symbol based on the values assigned

to it, it may still require type annotations in some cases. Pyright supports advanced features such as module-level type checking and can handle complex type relationships such as covariance and contravariance.

3. Pyre [7]: Pyre is a performant type checker for Python 3 that can operate on both annotated and unannotated code. Pyre uses abstract interpretation to infer types and can handle complex type systems such as generics and union types. Pyre also supports cross-file type checking and can be integrated with popular text editors such as VSCode.

4. Jedi [8]: Jedi is a Python autocompletion library that also provides type inference capabilities. Jedi can analyze a codebase to determine the types of variables, functions, and other constructs, and can also provide autocompletion suggestions based on this analysis. Jedi can handle complex code structures such as decorators and metaclasses.

5. Scalpel [13]: Scalpel is a static analysis framework that can infer the type information of all variables, including function return values and function parameters, in a Python program. It uses backward data-flow analysis and a set of heuristic rules to achieve high precision.

6. Pytype [9]: Pytype is a static type inference and checker for Python that uses type inference and flow analysis to determine the types of variables and expressions in a program. Pytype is designed to be fast and scalable and can be used to check the types of large codebases. Pytype uses a combination of type heuristics and abstract interpretation to infer types and can handle complex code structures such as decorators and metaclasses. It also supports type annotations in Python 3.

7. PySonar [14]: PySonar is a type inference tool for Python that uses flow analysis to determine the types of variables and expressions in a program. PySonar can be used to provide code completion and analysis in text editors and IDEs, and it can also be used to generate type hints for Python code. It can infer types for unannotated code and supports a variety of advanced features such as lambda functions and metaclasses.

While these tools provide valuable type inference capabilities for Python developers, there are some limitations to their use. Tools that require type annotations may not be suitable for legacy codebases that do not have type annotations and may require a significant amount of work to retrofit with annotations. Additionally, the quality of type inference depends heavily on the quality of type annotations, which may not always be accurate or comprehensive.

Tools that do not require type annotations may have limited accuracy, particularly for complex code structures or large codebases. In addition, the performance of these tools can vary widely depending on the codebase and the types of constructs used. Finally, these tools may not be suitable for large or complex codebases due to limitations in memory or processing power. Overall, these limitations highlight the need for continued development and improvement of type inference tools for Python.

2.2 Real-world benchmarks:

To evaluate the performance of type inference tools, a benchmark framework is needed that incorporates both micro-benchmarks and real-world benchmarks. The micro-benchmarks focus on specific language features and constructs, while the real-world benchmarks cover a wide range of Python code. Some popular real-world benchmarks include the ManyTypes4Py and Type4Py datasets, the TypePY dataset from PYInfer, the Typilus benchmark suite, and the OSS dataset of Typewriter.

1. ManyTypes4Py [11]: ManyTypes4Py is a large dataset designed to train machine learning models for predicting type annotations in Python. The dataset contains over 5,382 Python projects with more than 869K type annotations, all with mypy as a dependency. ManyTypes4Py includes the LibSA4Py tool, a light-weight static analyzer pipeline that processes Python projects and extracts type hints/features for training ML-based type inference models.

2. Type4Py [10]: Type4Py is a dataset that contains over 5.2K Python projects with 869K type annotations. The use of Pyre to generate additional annotations makes the dataset even more valuable, as it allows researchers to test the effectiveness of different type inference approaches.

3. PYInfer - typePY [1]: PYInfer - typePY is a benchmark suite designed to test the accuracy and performance of type inference tools. It consists of a set of Python programs with varying levels of complexity and type annotations. It contains 11 basic types and 500 common types and also includes user defined types

4. Typilus [4]: Typilus is a benchmark for type inference tools that focuses on generating test cases that are difficult to infer. It includes a range of Python programs that contain complex type dependencies and are challenging for type inference tools to handle.

5. OSS dataset of Typewriter [4]: The OSS dataset of Typewriter is a benchmark suite that is a collection of 1,137 popular open-source projects with 50+ stars. It is designed to evaluate the effectiveness of type inference tools on large-scale, real-world codebases.

Developers need to be able to evaluate the performance of type inference tools accurately to choose the most appropriate tool for their specific use case. However, there is currently no standard benchmark framework available for evaluating the performance of type inference tools in Python. The lack of a standard benchmark framework makes it challenging for developers to compare the performance of different type inference tools effectively. This issue can result in developers using suboptimal tools, which can lead to poor program performance and errors.

3

Problem Description

Type inference is a technique used in static analysis to determine the types of variables and expressions in a program without running the program. This information can be used to detect type-related errors, improve program comprehension, and enable other program transformations. The performance of type inference tools varies significantly based on the code's structure, the type of variables used, and other factors.

Despite the importance of type inference, there is currently a lack of standardized benchmarks for evaluating the performance and accuracy of type inference algorithms. This lack of standardized benchmarks makes it difficult for researchers and practitioners to compare the performance of different algorithms and to measure progress in the field. To address this issue, researchers have developed several real-world benchmarks for type inference[11, 10, 1, 3, 4]. These benchmarks aim to provide a more realistic and representative measure of the performance of type inference tools. However, these benchmarks vary significantly in their scope, methodology, and code they cover. For example, existing benchmarks were made to test out specific tools. The methodology of selecting projects varies significantly from checking mypy as a dependency to a ranking by GitHub stars. This makes it difficult to compare the results of different benchmarks or to draw meaningful conclusions about the relative performance of different type inference tools. Moreover, they might have failed to capture all the complexities of Python program. To address this problem, there is a need for a common real-world benchmark that covers a broad range of language features. This benchmark should be designed to be representative of real-world programs and should be large enough to provide a meaningful evaluation of performance and accuracy.

Another problem with the current state of type inference benchmarks is that there is no single micro-benchmark that covers all aspects of type inference. The existing benchmarks were either created for other functionalities, for instance, [15] for evaluating call graph construction, or is not robust enough to evaluate every feature [13]. As a result, it becomes difficult to evaluate the performance of type inference algorithms on a wide range of scenarios, which can lead to incomplete or misleading results. Hence there is a need to extend the benchmarks to cover type inference and include type annotations. Each module of the benchmark should test a specific aspect of type inference, and the modules should be easily configurable and combinable. This would allow researchers and practitioners to customize the benchmark according to their needs and evaluate specific aspects of type inference algorithms in isolation or combination.

The lack of a standardized framework for evaluating type inference algorithms presents a significant challenge for researchers and practitioners in the field. Without a common framework, it is difficult to compare the performance and accuracy of different tools and to build on

previous research. Hence, there is a need for a common framework for evaluating type inference algorithms. This framework should provide a standardized way of running benchmarks, collecting results, and comparing the performance and accuracy of different tools. Such a framework would enable researchers and practitioners to more easily reproduce experiments, share results, and build on each other's work. The framework should be flexible enough to accommodate different types of benchmarks and algorithms, but also standardized enough to ensure consistency across experiments.

To summarize, the lack of standardized benchmarks and common frameworks for evaluating type inference algorithms is a significant problem in the field. These shortcomings make it difficult to compare the performance of different algorithms and to measure progress in the field. To address these problems, there is a need for a comprehensive micro benchmark, a common real-world benchmark, and a common framework for evaluating type inference algorithms.

4 Thesis Goals

The primary objective of this thesis is to provide a benchmark framework that can be used by developers to evaluate type inference tools for Python. To achieve this objective, the following four main goals will be pursued:

4.1 Developing a Micro-benchmark

The first goal is to develop a micro-benchmark for evaluating the performance of specific language features and constructs in Python. This will involve identifying a set of commonly used language features and constructs and developing code snippets to test them. The micro-benchmark will also need to be designed in a way that can be easily extended to incorporate new language features and constructs as needed.

4.1.1 Diverse benchmark

The first goal of this research is to create a more comprehensive and diverse benchmark suite that covers a broad range of Python language features and constructs. This is essential to test the performance of type inference tools in different scenarios and to ensure that the tools can handle a wide variety of code snippets.

To achieve this goal, two existing benchmark suites will be combined: PyCG's [15] benchmark suite and Scalpel's [13] benchmark suite. PyCG's benchmark suite contains a comprehensive set of code snippets categorized into sub-directories based on the type of language feature, while Scalpel's benchmark suite includes a smaller set of test cases with ground truths for expected output. By integrating these two benchmark suites, a larger and more diverse benchmark suite can be created that covers a wide range of language features.

To combine the benchmark suites, the Scalpel test cases will first be organized into appropriate sub-directories within the PyCG suite based on their language feature. Next, the PyCG suite will be reviewed to identify any gaps in the coverage of language features, and additional test cases will be constructed to fill those gaps. The result will be a comprehensive benchmark suite that covers a broad range of Python language features and constructs.

4.1.2 Develop ground truths

Once the benchmark code snippets have been categorized, the next step is to develop the ground truth data for each benchmark test case. The ground truth data is essential to evaluate the

performance of each type of inference tool being tested. When a tool's inferred types match the expected output types, it will be deemed successful in correctly inferring the types for that code snippet.

Scalpel benchmark suite provides ground truths for each test case with their expected output types. To extend this to the combined benchmark suite, ground truths must be generated for each code snippet. The process involves using a type inference tool to generate a set of inferred types for each code snippet and manually converting the results into the required format. The resulting ground truth data will enable a standardized evaluation of the performance of different types of inference tools, making it easier to compare their results and identify areas for improvement.

4.1.3 Type annotations

In addition to the development of ground truths, the micro-benchmark will include type annotations to aid in proper type inference by various tools. As Python is a dynamically typed language, the use of type annotations is a relatively new feature, introduced in recent versions of the language. Type annotations will be utilized to specify expected variable and function argument types in the code.

Type annotation will be manually added for each variable and function argument in the code snippets. The annotations will be formatted in a manner that can be easily parsed and interpreted by the type inference tools being evaluated, following the new convention of Python coding, specifically PEP 484. Maintaining a copy of each code snippet with type annotations added will provide an additional option for tools that require type annotations to infer types. This will ensure that each tool is evaluated fairly and accurately, regardless of its ability to infer types without annotations.

4.2 Real-world benchmarks

The second goal is to standardise existing real-world benchmarks and extend them to cover a wide range of Python code. This will involve identifying a set of real-world Python programs and developing test cases to evaluate the performance of type inference tools on these programs.

4.2.1 Aggregated benchmarks

One way to aggregate these benchmarks is by utilizing existing datasets, such as ManyTypes4Py, Type4Py, PYInfer, Typilus, and TypeWriter. These datasets have been specifically curated to include a large number of Python programs with type annotations and are therefore ideal for evaluating the performance of type inference tools. For example, the ManyTypes4Py dataset contains over 5,000 Python projects with more than 869K type annotations, all of which have mypy as a dependency. The Type4Py dataset includes over 5,000 Python projects with 869K type annotations and used Pyre for type inference, resulting in 3.3 million type annotations.

4.2.2 Optional: Annotated real-world benchmark

Creating a standard benchmark for type inference tools requires a significant amount of effort to annotate real-world projects manually. Although annotation tools have improved, they might not always capture all the type information accurately, making manual annotation necessary. Manual annotation can capture subtle and context-specific type information that automated tools may overlook, leading to more precise and accurate type inference, which is crucial for some real-world projects.

4.3 Common Evaluation Framework

The third goal of this research is to develop a common evaluation framework that integrates both the micro-benchmark and real-world benchmarks to provide a standardized way of evaluating the performance of type inference tools. This framework will include a common input and result format for reporting the results of the evaluations.

4.3.1 Integration of Benchmarks

The initial step in creating a framework for evaluating type inference tools is to merge the micro-benchmark and real-world benchmarks. This framework will assess the performance of these tools across various Python language features and structures.

The goal is to construct a user-friendly interface that allows for the selection of either benchmark for evaluating type inference tools. The framework should smoothly integrate both benchmarks, ensuring an efficient and transparent testing process. This will enable users to choose the most fitting benchmark for their particular requirements and assess their tools' performance accordingly. The interface should also provide detailed information about the benchmark and evaluation results.

4.3.2 Common Input and Result Format

The second sub-goal is to develop a common input and result format for reporting the results of the evaluations. The input format will define how the code snippets are presented to the type inference tools, while the result format will define how the output of the tools is reported. Developing a common format will ensure that the evaluation results are consistent and can be easily compared across different tools.

Designing a common input format for type inference tools can be difficult since each tool may have different input requirements. Thus, creating a flexible input format that can adapt to the different requirements of various tools is a crucial sub-goal of the common framework development. However, it may be possible that some tools may not fit into the common input format, and the framework may need to be limited to the requirements of existing tools. In such cases, future tools may need to adapt to the established input format of the framework.

The common result format for reporting the evaluation results of the type inference tools will include various information such as the success rate, inferred types, any issues or errors with the tool, and the time taken for inference. By providing this information, the framework will enable users to compare the performance of different type inference tools and make informed decisions about which tool is best suited to their particular use case. Furthermore, the standardized result format will make it easier to reproduce and share the results of evaluations across different research groups and projects.

Overall, the development of a common evaluation framework will provide a standardized way of evaluating the performance of type inference tools and enable a comprehensive comparison of their results. The use of a common input and result format will ensure that the evaluation results are consistent and can be easily compared across different tools.

4.4 Evaluation

4.4.1 Evaluation of tools

Evaluate the performance of existing type inference tools using the common framework. Based on the goals of the research, some potential research questions for the evaluation are :

1. How do existing type inference tools for Python perform on the developed micro-benchmark?
   (a) Does deep learning-based tools support complex python features?
2. How does existing tools perform on the standardized real-world benchmark?
   (a) Does existing real-world benchmarks capture complex features of Python?

The evaluation and comparison of type inference tools will be done using the following metrics:

1. Exact Match: This metric measures the percentage of times the tool's prediction exactly matches the ground truth.
2. Base Type Match: This metric measures the percentage of times the tool's prediction matches the base type of the ground truth, ignoring type parameters. For example, List[str] and List[int] would be considered a match.
3. Rank-based Match: A rank-based metric can be employed to evaluate type inference tools, which rank the predicted types' complexity or rarity. This metric measures the percentage of times the tool's prediction is within a certain rank of the ground truth type, based on the criterion of how common or rare a type is. For instance, types seen more than a certain number of times in the training set can be categorized as common, while those seen less frequently can be considered rare. The metric can then calculate the percentage of predictions that fall within a certain rank of the ground truth type.

For each of the high level metrics precision and recall will be computed to compare the performance of the tools.

Moreover, completeness and soundness will be calculated for the evaluation of tools based on micro-benchmark.

1. Soundness refers to the ability of the type inference tool to infer all the types that can be determined from the program. A sound type inference tool should be able to infer the types of all variables and expressions in the program or in simple terms - no false negatives exists
2. Completeness on the other hand, refers to the ability of the type inference tool to infer only correct types. A complete type inference tool should not infer any incorrect types, meaning that all of its inferences should be accurate or in simple terms - no false positives.

4.4.2 Guidelines

Create a set of guidelines to help users utilize the common evaluation framework to assess the performance of type inference tools. These guidelines will be developed based on the insights gained from evaluating the performance of existing tools using the common framework. They will aim to provide clear instructions on how to use the framework effectively and interpret the evaluation results accurately.

Overall, the thesis work will provide a comprehensive and standardized benchmark framework for evaluating type inference tools for Python. The micro-benchmark and real-world benchmarks will cover a wide range of language features and constructs, and the common framework will provide a standardized way of evaluating the performance of type inference tools. The guidelines developed as part of the thesis work will also help developers effectively use the benchmark framework to evaluate the performance of type inference tools.

5 Thesis Structure

1. Introduction
   (a) Motivation
   (b) Problem Statement
   (c) Goals
2. Related Work
3. Problem Analysis
4. Methodology
   (a) Micro Benchmark
      i. Diverse benchmark
      ii. Ground truths
      iii. Type annotations
   (b) Real World Benchmark
   (c) Common Evaluation Framework
      i. Integration of Benchmarks
      ii. Common Input and Result Format
5. Implementation
6. Evaluation
7. Conclusion and Future Work
   (a) Limitations
   (b) Future work
8. Abbreviations
9. Bibliography

6 Time Plan

Task	Weeks
Micro Benchmark	
Diverse benchmark	1
Develop Ground Truths	3
Type Annotations	2
Real World Benchmark	
Integration	3
Common Framework	
Integration	1
Common input/output format	2
Evaluation	
Evaluating tools	2
Thesis Report	
Writing Thesis Report	6
Timeline Overview	
Total	20

Bibliography

[1] Siwei Cui, Gang Zhao, Zeyu Dai, Luochao Wang, Ruihong Huang, and Jeff Huang. Pyinfer: Deep learning semantic type inference for python variables. *CoRR*, abs/2106.14316, 2021.

[2] Yun Peng, Cuiyun Gao, Zongjie Li, Bowei Gao, David Lo, Qirun Zhang, and Michael Lyu. Static inference meets deep learning: A hybrid type inference approach for python. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, page 2019–2030, New York, NY, USA, 2022. Association for Computing Machinery.

[3] Miltiadis Allamanis, Earl T. Barr, Soline Ducousso, and Zheng Gao. Typilus: Neural type hints. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 91–105, New York, NY, USA, 2020. Association for Computing Machinery.

[4] Michael Pradel, Georgios Gousios, Jason Liu, and Satish Chandra. Typewriter: Neural type prediction with search-based validation. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, page 209–220, New York, NY, USA, 2020. Association for Computing Machinery.

[5] mypy - Optional Static Typing for Python — mypy-lang.org. `https://mypy-lang.org/`. [Accessed 06-Mar-2023].

[6] GitHub - microsoft/pyright: Static type checker for Python — github.com. `https://github.com/microsoft/pyright`. [Accessed 06-Mar-2023].

[7] Pyre | Pyre — pyre-check.org. `https://pyre-check.org/`. [Accessed 06-Mar-2023].

[8] Jedi - an awesome autocompletion, static analysis and refactoring library for Python 2014; Jedi 0.18.2 documentation — jedi.readthedocs.io. `https://jedi.readthedocs.io/en/latest/`. [Accessed 06-Mar-2023].

[9] GitHub - google/pytype: A static type analyzer for Python code — github.com. `https://github.com/google/pytype`. [Accessed 06-Mar-2023].

[10] Amir M. Mir, Evaldas Latoškinas, Sebastian Proksch, and Georgios Gousios. Type4py: Practical deep similarity learning-based type inference for python. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, page 2241–2252, New York, NY, USA, 2022. Association for Computing Machinery.

[11] Amir M. Mir, Evaldas Latoškinas, and Georgios Gousios. Manytypes4py: A benchmark python dataset for machine learning-based type inference. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 585–589, 2021.

[12] TypeT5: Seq2seq Type Inference using Static Analysis — arxiv.org. https://arxiv.org/abs/2303.09564. [Accessed 30-Mar-2023].

[13] Li Li, Jiawei Wang, and Haowei Quan. Scalpel: The python static analysis framework, 2022.

[14] GitHub - yinwang0/pysonar2: PySonar2: a semantic indexer for Python with interprocedual type inference — github.com. https://github.com/yinwang0/pysonar2. [Accessed 06-Mar-2023].

[15] Vitalis Salis, Thodoris Sotiropoulos, Panos Louridas, Diomidis Spinellis, and Dimitris Mitropoulos. Pycg: Practical call graph generation in python. In *Proceedings of the 43rd International Conference on Software Engineering*, ICSE '21, page 1646–1657. IEEE Press, 2021.