TypeEvalPy: A Micro-benchmarking Framework for Python Type Inference Tools

Ashwin Prasad S. Venkatesh[§], Samkutty Sabu[¶], Jiawei Wang[†], Amir M. Mir[‡], Li Li[*], Eric Bodden[**]
[§] *ashwin.prasad@upb.de*, Heinz Nixdorf Institut, Paderborn University, Paderborn, Germany
[¶] *samkutty@mail.uni-paderborn.de*, Paderborn University, Paderborn, Germany
[†] *jiawei.wang1@monash.edu*, Faculty of Information Technology, Monash University, Melbourne, Australia
[‡] *s.a.m.mir@tudelft.nl*, Delft University of Technology, Delft, The Netherlands
[*] *lilicoding@ieee.org*, School of Software, Beihang University, Beijing, China
[**] *eric.bodden@upb.de*, Heinz Nixdorf Institut & Fraunhofer IEM, Paderborn University, Paderborn, Germany

ABSTRACT

In light of the growing interest in type inference research for Python, both researchers and practitioners require a standardized process to assess the performance of various type inference techniques. This paper introduces TypeEvalPy, a comprehensive micro-benchmarking framework for evaluating type inference tools. TypeEvalPy contains 154 code snippets with 845 type annotations across 18 categories that target various Python features. The framework manages the execution of containerized tools, transforms inferred types into a standardized format, and produces meaningful metrics for assessment. Through our analysis, we compare the performance of six type inference tools, highlighting their strengths and limitations. Our findings provide a foundation for further research and optimization in the domain of Python type inference.

ACM Reference Format:
Ashwin Prasad S. Venkatesh[§], Samkutty Sabu[¶], Jiawei Wang[†], Amir M. Mir[‡], Li Li[*], Eric Bodden[**]. 2024. TypeEvalPy: A Micro-benchmarking Framework for Python Type Inference Tools. In *2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3639478.3640033

1 INTRODUCTION

Type inference refers to the process of automatically determining the data type of an expression within a programming language. In Python, which is dynamically typed, this determination takes place at runtime. To address potential ambiguities, developers can utilize type annotations, which explicitly specifies the expected data types of variables or function returns. As the complexity of software projects increases, programmers find it increasingly challenging to maintain consistent data types. In response to this challenge, both industry and academia have developed type inference tools and static type checkers. Examples from industry include *Pyright* [1] and *Pytype* [3], while academic contributions feature *Type4Py* [7] and *HiTyper* [8]. The topic of type inference in Python is a growing area of research and tool development within the software engineering community. Current efforts focus on understanding the advantages of enforcing type annotations, and on finding ways to infer types in Python code that lacks type annotations.

In recent years, many solutions for type inference have been proposed. However, a unified and comprehensive evaluation framework for these tools is still lacking. Current literature primarily assesses the performance of such type inference tools based on large-scale real-world benchmark datasets, notably *Type4Py*, *HiTyper*, and *Typilus* [4]. On the contrary, open-sourced solutions only rely on specifically-designed test cases. This evaluation approach, nonetheless, presents several limitations: (1) Different studies might report findings based on different datasets, complicating a direct comparison and understanding of the relative merits and drawbacks of each tool. (2) Type annotations in real-world datasets are sometimes erroneous. (3) Evaluations often provide a broad-brush score, overlooking nuanced insights into specific technical challenges, including the treatment of diverse language constructs.

In this paper, we introduce TypeEvalPy, a type inference evaluation framework for Python bundled with a micro-benchmark that covers all the Python language constructs of Python 3.10.

Our primary objective with TypeEvalPy is to provide insights into the recent advances in type inference tooling for Python programs. When presented with an executable type inference tool, TypeEvalPy processes the tool with input from the built-in micro-benchmark and outputs the inferred type information in a standard format for further analysis. Then, TypeEvalPy analyzes the output of each tool and reports the comparative analysis using a set of metrics such as exact match rate, precision, etc (c.f. section 2.3).

We demonstrate TypeEvalPy's utility by evaluating six state-of-the-art type inference tools, including two ML-based and four static analysis-based approaches. For ML-based approaches, we extend our analysis to incorporate top-*n* predictions. Our empirical findings reveal that the performance of type inference can be significantly enhanced by integrating external user-annotated type stubs and combining static analysis with ML techniques. The state-of-the-art hybrid strategy in *HiTyper*, outperforms its purely static analysis based counterpart. However, the underlying static analysis technique that *HiTyper* uses, performs poorly compared to the other pure static analysis-based alternatives, indicating that the performance of *HiTyper* can be improved. Moreover, we posit that researchers should place greater emphasis on function parameter annotations, particularly since the majority of tools generate only a limited number of these annotations. Nevertheless, achieving

ICSE-Companion '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0502-1/24/04.
https://doi.org/10.1145/3639478.3640033

soundness in type inference remains a challenge, with even the top-performing tools achieving a soundness rate of only 44%.

The paper is organized as follows: The technical design is detailed in section 2 followed by experimental results being reported in section 3. After a brief discussion about the results in section 4, the paper is summarized in section 5.

Availability. TypeEvalPy is published on GitHub as open-source software: https://github.com/secure-software-engineering/TypeEvalPy

2 TYPEEVALPY FRAMEWORK

The primary goal of TypeEvalPy is to offer a comprehensive, standardized, and reproducible benchmarking system for evaluating type inference tools in Python. To this end, TypeEvalPy contains a diverse set of 154 code snippets with 845 type annotations across 18 categories that capture the nuances of various Python features. For a tool to be benchmarked using TypeEvalPy, it must be adapted into a containerized format that aligns with TypeEvalPy's specifications. To simplify this adaptation process, we are sharing a template. For reference, TypeEvalPy already includes containerized versions of six type inference tools: *HeaderGen* [10], *Jedi* [2], *pyright* [1], *Scalpel* [6], *HiTyper* [8], and *Type4Py* [7].

The TypeEvalPy framework is organized into three main modules: `Runner`, `Translator`, and `Result Analyzer`. First, the `runner` module manages the initiation and execution of containerized tools, specifically running type inference on the micro-benchmark. Following this, the `translator` module takes on the role of transforming the inferred types into a standardized format, making them suitable for comparison. Lastly, the `analyzer` compiles the results from all the tools and provides meaningful metrics for assessment. Importantly, the modules realize a high degree of automation, thereby ensuring the reproducibility of results in the academic context.

2.1 Micro-benchmark

The micro-benchmark of TypeEvalPy comprises of 154 program snippets containing 845 type annotations. To ensure a comprehensive coverage of Python's language features, snippets are subdivided into 18 categories, each representing a specific feature. This categorization is based on PyCG's [9] call graph benchmark and we extend it to address gaps in the coverage of language features. Our strategy for enhancing the benchmark involved inspecting the Python manual to ensure all language features are adequately represented. Furthermore, we categorized test cases from existing inference tools, ensuring the comprehensiveness of our benchmark. Additionally, the micro-benchmark's modular design allows for easy extensions. The micro-benchmark consists of the following categories: *args (8)*, *assignments (8)*, *builtins (7)*, *classes (26)*, *decorators (8)*, *dicts (15)*, *direct_calls (6)*, *dynamic (3)*, *exceptions (2)*, *external (7)*, *functions (9)*, *generators (6)*, *imports (14)*, *kwargs (4)*, *lambdas (6)*, *lists (10)*, *mro (7)*, *returns (8)*.

Type annotation format is based on the Scalpel framework [6] and are stored as JSON files with the code snippets. They contain the following: (1) *file*: denotes the filename, (2) *line_number:* specific line in the file. (3) *col_offset:* represents the indentation. (4) *type:* list of types. (5) *function:* function name, if the annotation is within one. (6) *variable:* specifies the variable's name being annotated. (7) *parameter:* captures the name of a function argument, if relevant.

Type annotations are categorized into three categories: (1) Function return (FR) type, (2) Function parameter (FP) type, and (3) Local variable (LV) type. In total, the micro-benchmark consists of 239 FR, 88 FP, and 518 LV type annotations. To construct the ground truth, the first two authors manually inspected each code snippet and, where required, used a debugger to verify the run-time type of each Python element. To further mitigate potential errors, each file was reviewed consecutively by both authors.

During the development of the ground truth, we made several design decisions: (1) Type annotations for generics are not concretized, for instance, a list of integers is annotated as `List` instead of `List[Int]`. (2) FPs are annotated based on their usage. In cases where a function can return multiple types, special care was taken to ensure that the function is called with parameters of all types the function supports. (3) FR types and local variables defined inside a function are context-insensitive, i.e., if a variable defined inside a function can take on multiple types based on different calling contexts, each variable is annotated with all possible types it can have during runtime. (4) More generally, types assigned to entities were chosen to reflect all the possible runtime types *in the given program*, and we chose the most specific type possible. Therefore, none of the type annotations are marked as "`Any`".

2.2 Runner and Translator

The primary responsibility of the `runner` module is to orchestrate the execution of containerized type inference tools on the micro-benchmark. For each type inference tool, the `runner` module creates an instance of the Docker container, the micro-benchmark is copied into the running instance and runs the type inference inside it. Then, the `runner` module uses the `translator` to convert results into the TypeEvalPy format. Once each tool finishes running, the `runner` module calls the `result analyser` module.

2.3 Result Analyzer

The `analyzer` module produces detailed statistics for comparing the effectiveness of different tools as listed below:

- **Exact matches:** The number of inferred types that exactly match the ground truth. This metric is used widely used in the literature to evaluate type inference tools [4, 7, 8].
- **Precision:** The fraction of reported types that are exactly inferred according to the ground truth.
- **Recall:** The number of actual types that are exactly reported by the type inference tool.
- **Soundness:** Whether the type inference tool identifies all possible types specified in the Python code to ensure none are omitted. Reported as a boolean per code snippet in the micro-benchmark.
- **Completeness:** Whether the tool accurately reports only the types that are present, avoiding any incorrect or extraneous types. Reported as a boolean per code snippet in the micro-benchmark.
- **Top-*n* prediction comparison:** The accuracy comparison of probabilistic tools when considering their top-*n* inferred types. This metric is widely used to evaluate ML predictors [5, 7, 8].
- **Report of missing types:** List of types that are present in the ground truth but are unreported by the tools.

- **Report of mismatched types:** List of types reported by the tools that do not align exactly with the ground truth.

3 EXPERIMENTS

We used the TypeEvalPy framework to evaluate the most recent versions of the following tools: HiTyper [8] and Type4Py [7] as examples of machine learning tools, Jedi [2] and Pyright [1] were chosen as open-source tools, while HeaderGen [10] and Scalpel [6] were picked as representations of academic tools.

HiTyper is a hybrid analysis approach that employs both static analysis and ML to infer types. It also has an option to only use static analysis for type inference, which is based on PyCG. We evaluated HiTyper in both modes. In discussions, the static-analysis method is referred to as "*HiTyper*," while the hybrid method is referred to as "*HiTyper-Dl*", which integrates Type4Py.

Results. Table 1 shows the exact matches of the selected tools for each type categories: (1) Function return (FR) type , (2) Function parameter (FP) type, and (3) Local variables (LV). The tools are arranged in the table according to their performance from left to right. HeaderGen performed the best with highest overall performance in all categories except in `builtins` and `external` categories. In these specific categories, both Jedi and Pyright performed the best. This suggests that Jedi and Pyright are more adept at integrating user-specified type hints, commonly referred to as *typestubs*. On the contrary, Jedi and Pyright fail to infer types of FPs. Both the tools are designed to infer function parameters as "`Any`" in most cases, except for cases where a function is passed as a parameter. This strategy, while congruent with Python's duck typing paradigm, limits the applicability of the inferred types in a wider context.

Among the ML tools, HiTyper-DL outperforms Type4Py with 369 exact matches, while Type4Py had 157 matches. Additionally, HiTyper-DL shows a notably improved performance compared to HiTyper, which relies solely on static analysis.

Soundness and Completeness. Table 1 lists the soundness and completeness values at the bottom. HeaderGen is sound in 68 of the 154 cases, i.e., it did not miss any types in these cases. HeaderGen is complete in 55 of the 154 cases, i.e., it did not falsely identify types in these cases. HeaderGen demonstrated the most balanced performance compared to all other tools. HiTyper has a low score of 3 in soundness, HiTyper-DL improves this score to 18. While this score is a modest increase, it is notably better than HiTyper. In the assessment of HiTyper, while its completeness score appears promising, it is noteworthy that the tool did not produce predictions for 34 out of the 154 cases. Furthermore, HiTyper often failed to infer types, as indicated by its soundness and exact matches.

top-*n* Matches. Table 2 shows top-*n* results of ML tools and its comparison with HeaderGen. HiTyper-DL shows significant improvement considering top-5 predictions with 441 exact matches which is 78.2% of HeaderGen's score. However, the difference between top-*n* values of 3 and 5 are small, indicating that the majority of correct predictions by HiTyper-DL fall within the top-3 types. Type4Py benefits immensely when top-2 and top-3 are taken into account, the exact matches nearly doubled the score from top-1 to top-3. However, similar to HiTyper-DL, the majority of the correct predictions are within the top-3 types. Overall, while ML-based tools demonstrate promise, they still trail behind the performance of HeaderGen when considering the top-1 predictions.

4 DISCUSSION

In this section, we highlight the outcomes for each tool, encapsulating their strengths and weaknesses as observed by TypeEvalPy:

HeaderGen. The analysis revealed that HeaderGen performed consistently across all categories. With flow-sensitive analysis built on top of PyCG, HeaderGen is able to infer types of Python elements accurately. However, it showed limitations in the `builtins` and `external` categories, highlighting the lack of support for analyzing external source code. The support for utilizing typestubs in analysis is limited, such as, support for typestubs with overriding definitions for the same function based on FP types is not implemented.

Jedi. The open-source community driven tool Jedi has been enhanced to address diverse challenges and to provide broad analysis capabilities. Specifically, its ability to reason about external source code in the `builtins` and `external` categories is vital for analyzing real-world code. However, Jedi's design choice to omit the output of types for FPs hinders its overall result. Furthermore, our analysis of the mismatch reports from TypeEvalPy highlighted an inconsistency. When a function is passed as reference in an argument to a function call, both Jedi and Pyright incorrectly infer the variable type as the return type of that function. In reality, the correct type should be `callable`, since the function is not actually called. In TypeEvalPy, Pyright and Jedi exhibited this behavior in 57 and 18 instances, respectively, highlighting the potential for improvement.

Pyright. Microsoft's Pyright demonstrated strong performance in the `builtins` and `external` categories. It also performed slightly better in the FP and LV categories than Jedi. It is also pertinent to mention that, Pyright, developed in TypeScript, lacks interfaces to access its internal analysis structures. This limitation makes general-purpose analysis difficult. To address this, we created a language server protocol (LSP) client in Python, allowing TypeEvalPy to query results for each element. The static analysis community can benefit from such an interface if its built into Pyright.

Scalpel. Scalpel exhibited strong performance in aspects related to functions. However, it does not currently support the output of LV types, which significantly impacted its overall ranking and needs improvement in this area. For instance, it can annotate up to 155 function return types and 32 function parameters, ranking second among all pure static analysis tools for the two categories. Regarding LV, Scalpel only provides six annotations at the second last position. Furthermore, it does not handle external library calls, which are common in real-world projects.

Type4Py. Type4Py uses a deep similarity learning-based technique, meaning that it can only infer types that were seen during the training phase. Therefore, Type4Py performs poorly compared to static approaches such as HeaderGen and Pyright. Also, Type4Py only learns from identifiers in the method signature and also the usage of FPs and LVs inside the method body. This may also explain why Type4Py fails to infer types for some of the categories in the microbenchmark, i.e., classes, dynamic, and exceptions. It is also worth mentioning that Type4Py is mostly trained on local variables data and hence it performs relatively better on LV types compared to FR and FP types. For better performance, top-5 suggestions from

Table 1: Comparison of exact matches, sound, and complete values of type inference tools for micro-benchmark categories

FR: Function return type, **FP:** Function parameter type, **LV:** Local variable type, **845:** Total type annotations, **154:** Total test cases

Category	HeaderGen			Jedi			Pyright			HiTyper-DL			HiTyper			Scalpel			Type4Py		
	FR	FP	LV	FR	FP	LV	FR	FP	LV	FR	FP	LV	FR	FP	LV	FR	FP	LV	FR	FP	LV
args	17	9	12	12	0	9	8	1	8	12	0	6	8	0	0	8	7	0	11	2	6
assignments	15	1	33	20	0	21	20	0	25	21	4	9	20	0	5	20	2	1	0	2	5
builtins	0	0	26	0	0	21	1	0	45	1	2	18	1	0	17	0	0	0	1	2	8
classes	39	7	67	0	0	57	1	0	46	27	2	41	24	0	23	25	0	0	0	0	17
decorators	11	6	2	10	0	8	7	0	3	8	0	3	7	0	0	16	3	0	7	0	3
dicts	23	3	60	21	0	34	19	2	50	20	3	22	20	2	16	19	2	1	2	3	18
direct_calls	10	3	8	6	0	7	3	0	6	3	2	4	2	0	0	5	1	0	2	2	4
dynamic	1	0	2	1	0	2	1	0	2	1	0	5	1	0	2	1	0	0	0	0	5
exceptions	0	0	2	0	0	1	0	0	1	0	0	1	0	0	1	0	0	0	0	0	0
external	0	0	3	0	0	8	0	0	2	1	0	4	0	0	2	0	1	0	1	0	1
functions	8	9	12	5	0	14	5	2	13	5	5	5	3	2	1	6	5	1	1	3	4
generators	9	4	17	5	0	23	4	3	18	10	5	11	10	3	11	6	1	3	1	1	3
imports	3	0	11	1	0	16	3	0	20	3	0	11	3	0	0	3	0	0	3	0	10
kwargs	8	5	5	7	0	4	4	0	5	7	0	0	4	0	0	4	4	0	3	0	0
lambdas	3	7	4	6	0	11	2	0	1	3	0	7	3	0	5	2	4	0	0	0	2
lists	14	1	26	17	0	27	13	0	25	16	3	16	13	0	13	16	1	0	2	3	4
mro	14	0	16	0	0	13	0	0	14	15	0	11	13	0	6	13	0	0	0	0	4
returns	11	1	16	11	0	17	9	0	13	10	1	5	9	0	0	11	1	0	5	1	5
Total	**186**	**56**	**322**	**122**	**0**	**293**	**100**	**8**	**297**	**163**	**27**	**179**	**141**	**7**	**102**	**155**	**32**	**6**	**39**	**19**	**99**
	564/845			**415/845**			**405/845**			**369/845**			**250/845**			**193/845**			**157/845**		
Sound	**68/154**			**24/154**			**21/154**			**18/154**			**3/154**			**0/154**			**5/154**		
Complete	**55/154**			**30/154**			**91/154**			**32/154**			**135/154**			**81/154**			**11/154**		

Table 2: top-*n* exact matches comparison with ML tools

Tool	top-*n*	FR	FP	LV	Total
HeaderGen	1	186	56	322	564
HiTyper-DL	1	163	27	179	369
	3	173	37	225	435
	5	175	37	229	441
Type4Py	1	39	19	99	157
	3	103	31	167	301
	5	109	31	174	314

Type4Py should be considered, providing that it performs *k*-nearest neighbor search to find possible type annotations for a given query.

HiTyper & HiTyper-DL. HiTyper is a hybrid type inference approach, which combines a deep learning model, i.e., Type4Py with static analysis. As expected, it performs better than Type4Py, a pure ML-based approach. In general, HiTyper's static inference part seems to be quite imprecise as it is very unsound but more complete than the other baselines. On the other hand, HiTyper-DL is more sound but incomplete. This can be explained by the fact that it uses type rejection rules to be more precise. However, HiTyper-DL can be expensive to run for large projects considering its hybrid nature.

Though showing promising results, the hybrid paradigm does not show many advances in soundness. Only 15 of 154 sound results are brought by the deep learning model, which, however, largely reduces the completeness by 103 code snippets.

Outlook. In our study, we found that HeaderGen performs reliably in several complex scenarios. Yet, for code that depends significantly on external libraries and has dependable user-defined type stubs, Pyright and Jedi seem more suitable due to their enhanced integration with type stubs. Additionally, the hybrid HiTyper-DL approach shows potential. Future research can explore how HeaderGen might be combined with HiTyper-DL to improve outcomes.

5 CONCLUSION

In this paper, we presented TypeEvalPy, a micro-benchmarking framework designed for assessing Python type inference tools. Our comprehensive analysis covered a diverse array of six type inference tools including static analysis based approaches, ML-based approaches, and hybrid approaches. Notably, HeaderGen performed the best in terms of exact matches, soundness, and completeness. Jedi and Pyright followed close to each other, ranking second and third, respectively. Moreover, HiTyper-DL, a hybrid type inference tool, demonstrated potential advantages over solely ML-based alternatives, securing the fourth position. The comparative insights from TypeEvalPy highlights the differences between these tools and sets the stage for future research and optimization endeavors. Overall, the challenge of type inference in Python remains unresolved, presenting opportunities for advancement in terms of both soundness and completeness.

ACKNOWLEDGMENTS

Funding for this study was provided by the Ministry of Culture and Science of the State of North Rhine-Westphalia under the SAIL project with the grand no NW21-059D.

REFERENCES

[1] [n. d.]. GitHub - microsoft/pyright: Static type checker for Python — github.com. https://github.com/microsoft/pyright.

[2] [n. d.]. Jedi - an awesome autocompletion, static analysis and refactoring library for Python. https://jedi.readthedocs.io/en/latest/.

[3] 2023. pytype: A type analyzer by Google. https://github.com/google/pytype

[4] Miltiadis Allamanis, Earl T. Barr, Soline Ducousso, and Zheng Gao. 2020. Typilus: Neural Type Hints *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 91–105. https://doi.org/10.1145/3385412.3385997

[5] Siwei Cui, Gang Zhao, Zeyu Dai, Luochao Wang, Ruihong Huang, and Jeff Huang. 2021. PYInfer: Deep Learning Semantic Type Inference for Python Variables. *CoRR* abs/2106.14316 (2021). arXiv:2106.14316 https://arxiv.org/abs/2106.14316

[6] Li Li, Jiawei Wang, and Haowei Quan. 2022. Scalpel: The Python Static Analysis Framework. https://doi.org/10.48550/ARXIV.2202.11840

[7] Amir M. Mir, Evaldas Latoškinas, Sebastian Proksch, and Georgios Gousios. 2022. Type4Py: Practical Deep Similarity Learning-Based Type Inference for Python. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 2241–2252. https://doi.org/10.1145/3510003.3510124

[8] Yun Peng, Cuiyun Gao, Zongjie Li, Bowei Gao, David Lo, Qirun Zhang, and Michael Lyu. 2022. Static Inference Meets Deep Learning: A Hybrid Type Inference Approach for Python. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. 2019–2030. https://doi.org/10.1145/3510003.3510038

[9] Vitalis Salis, Thodoris Sotiropoulos, Panos Louridas, Diomidis Spinellis, and Dimitris Mitropoulos. 2021. PyCG: Practical Call Graph Generation in Python. In *Proceedings of the 43rd International Conference on Software Engineering* (Madrid, Spain) *(ICSE '21)*. 1646–1657. https://doi.org/10.1109/ICSE43902.2021.00146

[10] Ashwin Prasad Shivarpatna Venkatesh, Jiawei Wang, Li Li, and Eric Bodden. 2023. Enhancing Comprehension and Navigation in Jupyter Notebooks with Static Analysis. IEEE Computer Society, 391–401. https://doi.org/10.1109/SANER56733.2023.00044