PyHintSearch: Validated Combinatorial Search for Probabilistic Type Inference Models

MSc Computer Science

Mark Bekooy

February 2024

PyHintSearch: Validated Combinatorial Search for Probabilistic Type Inference Models

by

Mark Bekooy

to obtain the degree of Master of Science

at the Delft University of Technology,

to be defended publicly on Tuesday March 5, 2024 at 3:00 PM.

Student number:	4650972	
Project duration:	April 21, 2023 – March 5, 2024	
Thesis committee:	Associate Prof. Dr. Georgios Gousios,	TU Delft, supervisor
	Assistant Prof. Dr. Casper Bach Poulsen,	TU Delft
	Assistant Prof. Dr. Sebastian Proksch,	TU Delft

Abstract

Type annotations in Python are an integral part of static analysis. They can be used for code documentation, error detection and the development of cleaner architectures. By enhancing code quality, they contribute to the robustness, maintainability and comprehensibility of codebases. Tools like static type checkers use type annotations to detect bugs early, with some type checkers like Pyright being capable of inferring annotations statically from source code.

This thesis uses an innovative approach to further enhance type annotation coverage in Python codebases by using a combination of machine learning predictions and combinatorial search. To do this, PyHintSearch was developed. PyHintSearch constructs a search tree to which a depth-first search is applied to systematically explore potential combinations of predicted type annotations and validate them using feedback from the Pyright static type checker. Ultimately, the goal is to identify a branch containing a valid combination of type annotations. These annotations can then be integrated into Python code, thereby enhancing the type annotation coverage, which leads to improved static analysis and ultimately better code quality.

PyHintSearch's effectiveness is evaluated based on type annotation coverage and correctness, performance, and practical usability. Experimental results demonstrate different improvements in type annotation coverage, depending on the machine learning model used for type inference. Type4Py showed an improvement of 62.45% and TypeT5 of 79.93%. The precision of type annotations from these models are 0.36 and 0.51, respectively. Performance-wise, PyHintSearch can efficiently explore the exponential search space, annotating 16 diverse projects, ranging from small to large, in approximately 13.75 hours when using the Type4Py model. Regarding practical usability, the impact of type annotations on downstream program analysis is examined through the generation of call graphs. The additional information that type annotations provide can be used to refine the call graph by eliminating irrelevant calls to make it more precise.

Preface

Studying Computer Science at the Delft University of Technology was one of the best choices I have made in my life. The years have flown by filled with challenging topics, thought-provoking ideas and a great atmosphere, together with plenty of extracurricular activities. This made studying in Delft a fantastic experience. Now the end of an era is near and I am delighted to present my master's thesis as a conclusion to this journey.

First of all, I would like to thank my supervisor Prof. Georgios Gousios for his invaluable guidance, encouragement and great discussions throughout my thesis journey. I really appreciated the weekly meetings and the time taken to answer my questions and review my work. It truly felt like a collaboration where we had a shared goal to achieve.

I also want to thank Prof. Casper Bach Poulsen and Prof. Sebastian Proksch for being part of my thesis committee. There are plenty of things to do during the day as a professor at a university, so I truly appreciate both of you taking the time to evaluate and grade my thesis.

Additionally, I want to thank Endor Labs for providing their proprietary callgraph generator for my thesis. It was very useful for the evaluation of my developed PyHintSearch tool, especially to test its practical applicability.

Furthermore, I would like to thank my friends for all their unconditional support. Thank you Daan den Ouden, Rolf Pieperbrink for being wonderful friends throughout the years of study and for the great lunches together. Thank you Peter Nelemans and Timon Bestebreur, for being fantastic friends with whom I could talk every two weeks, sharing our experiences and thoughts. Thanks to the members of my student association, C.S.R. Delft, for all the great conversations and checking in on the progress of my research. Thank you to my housemates, Hanno Blankenstijn and Mirte Brouwer, for literally always being there, our fantastic discussions, keeping me happy and much much more.

Likewise, a huge thanks to my parents and my sister who supported me throughout this project. You always cheered for me, but also gave me time to breathe and relax a little.

Finally, I want to thank God. He helped me throughout my studies, my thesis and many more aspects of my life. He is the one who can always be trusted, gave me moments of peace when I needed them most, and is the source of my enthusiasm and optimistic worldview.

Mark Bekooy
Delft, February 2024

Contents

1
Introduction

Code quality is important in software development, as it directly impacts the robustness, maintainability and comprehensibility of a codebase. The higher the quality of the code, the lower the number of errors that occur, leading to reduced costs for debugging and maintenance. This is particularly critical for large tech companies such as Microsoft, Google, Facebook, Amazon, etc. whose codebases consist of millions of lines of code. For these companies, maintaining well-organised and error-resistant codebases is essential for delivering high-quality and uninterrupted services to their users.

Many of these companies use Python in their codebases [21], for things like web development and data science, because of its simplicity, readability and extensive library support. Because of its versatility, Python has become extremely common in modern software development. In fact, according to the Stack Overflow Developer Survey 2023 [27] and GitHub Octoverse 2023 [7], Python is one of the most popular programming languages in the world, outranking Java and C/C++.

However, Python development has a challenge concerning type safety and reliability. As a dynamically-typed language, Python allows variable types to change during the runtime of a program, potentially leading to unexpected behaviour and crashes. Type annotations offer a solution for this by explicitly declaring the expected types for variables, function parameters and return values. Integrating these annotations improves code maintenance, comprehension and error detection, thereby elevating code quality. However, despite their importance, many Python projects lack sufficient type annotations [15].

To address this challenge, this thesis focuses on enhancing the type annotation coverage of Python code through a validated combinatorial search for probabilistic type inference models. By incorporating additional type annotations, static analysis has more information available, enabling more accurate error detection during the development process and contributing to increased code quality and reliability. These improvements lead to a smoother development workflow, reduced debugging efforts and improved software reliability, ultimately benefiting both developers and end-users of products and services.

1.1. Type Annotations in Python

Python's dynamic typing allows for rapid development and flexible coding styles, but it can make understanding and maintaining code more challenging, especially in larger projects. To address this, Python Enhancement Proposal (PEP) 484 [35] was implemented in Python version 3.5. It introduced type annotations, which can be gradually integrated into existing Python code. Gradual typing [41] enables programmers to mix typed and untyped code by incrementally adding type annotations and choosing the level of type safety they want to use.

These type annotations, which can also be referred to as type hints, are optional in Python and do not enforce strict typing as Python remains dynamically typed. Instead, they serve as a form of documentation and can be used by tools like linters and static type checkers to catch potential errors early in the development process. Furthermore, they provide information to Integrated Development Environments (IDEs) for syntax highlighting and support during code development, and they can play a similar role as tests [4, 11].

For a visual understanding of type annotations, see Figures 1.1a and 1.1b. Figure 1.1a illustrates a straightforward Python function that adds two numbers and Figure 1.1b presents an identical example that includes type annotations.

```python
1  def add_numbers(x, y):
2      addition = x + y
3      return addition
```

(a) Without type annotations.

```python
1  def add_numbers(x: int, y: int) -> int:
2      addition: int = x + y
3      return addition
```

(b) With type annotations.

Figure 1.1: Identical Python addition function.

As can be seen in line 1 of Figure 1.1b, type annotations serve the purpose of indicating the types of parameters and the return type of a function. Additionally, as shown in line 2, the type of the variable in the statement can also be explicitly specified. This designated area where a type annotation can be added is commonly referred to as a “type slot”. Filling these type slots is a tedious and time-consuming task. While annotating programs is considered a good practice, automating this process would save developers valuable time and effort. To address this challenge, various tools and models have been developed to automatically infer type annotations.

1.2. Problem Statement

Static type checkers like Pyre [9] and Pyright [24] can infer type annotations from source code, but they often fall short in determining all available type slots. To address this gap, machine learning models offer a solution by predicting type annotations for the remaining slots, enhancing the type annotation coverage of Python code. However, since these models can predict multiple annotations for the same type slot, it introduces a challenge. Namely, selecting the highest-rated prediction (top-1) for each type slot may lead to a combination of annotations that does not fit in the code context. Therefore, exploring different combinations of predicted type annotations is crucial for finding a valid combination that does fit in the context. Using more predictions (top-3 or top-5) allows for more exploration, but also leads to an exponential growth of the search space. Navigating this expanded search space takes significantly longer due to the increased number of combinations that need to be checked. For that reason, a fast search strategy is needed to efficiently validate type combinations based on feedback from a static type checker.

1.3. Research Questions

The primary objective of this thesis is to identify a valid combination of machine learning predicted type annotations, building upon the Pyright-inferred types. This objective is pursued through the implementation of a combinatorial search strategy. For this, the PyHintSearch tool was developed, which evaluates predicted type annotations by leveraging Pyright's type-checking capabilities and utilising its feedback for validation. To assess the effectiveness of PyHintSearch, several questions arise. First of all, this thesis aims to investigate the additional number of annotations achievable. This leads to the first research question:

- **RQ1 (Coverage): What is the overall increase in the number of type annotations when combining static type inference with a machine learning model?**

Moreover, PyHintSearch must be practically usable. Therefore, not only must the combinatorial search be effective in identifying a valid combination of type annotations in an exponential search space, but it must also be efficient since long waiting times are undesirable. The faster the exploration of this search space, the quicker the automation of adding type annotations becomes. Additionally, understanding the tool's memory usage provides insight into the necessary resources that are required for its operation. This brings forth the second research question:

- **RQ2 (Performance): What are the performance characteristics, in terms of speed, memory usage and scalability, of the PyHintSearch tool?**

Lastly, quickly adding many accurate type annotations is beneficial. However, aside from their value in documentation and bug detection, their potential influence on other aspects of a program also raises questions. For instance, do type annotations have an impact on the functionality of a program? And if so, how? Based on this, the last research question is defined as:

- **RQ3 (Use case): How does the presence of type annotations impact downstream program analysis?**

1.4. Main Contributions

The main contributions of this thesis revolve around a search-based validation strategy aimed at efficiently exploring the exponential space of potential type annotation combinations. These contributions are summarised as follows:

1. **Combinatorial Search Strategy** The first contribution entails a search technique that systematically explores numerous potential combinations of predicted type annotations to identify a final valid combination.

2. **PyHintSearch** The second contribution is the implementation of PyHintSearch, a comprehensive tool that integrates the aforementioned combinatorial search strategy with various practical components to enhance the process of type annotating Python codebases.

1.5. Outline

The structure of this thesis is as follows: Chapter 2 provides background information about static analysis and type inference topics. Chapter 3 dives into various machine learning type inference models examined in related work. Chapter 4 details the inner workings of the combinatorial search, the components of the developed tool and the used evaluation strategy. Next, Chapter 5 presents the results regarding type annotation coverage, tool performance and the practical usage of type annotations for generating call graphs for Python code. Chapter 6 then discusses the main findings and limitations of this thesis and, finally, Chapter 7 concludes this thesis and suggests directions for future research.

2 Background

This background chapter gives an overview of several core concepts, laying the foundation for a comprehensive understanding of topics related to the research problem. It discusses topics related to static analysis, the difference between statically-typed and dynamically-typed languages, classical type inference, and the growing role of machine learning in type inference. Additionally, it explores call graphs and their utility in program analysis.

2.1. Static Analysis

Static analysis is a software development practice that involves examining source code or compiled code without executing it. It aims to uncover potential errors, vulnerabilities and other issues that could lead to bugs or security vulnerabilities in the software. It does this by analysing code syntax, structure and dependencies to identify potential problems [3, 13]. Although developers can manually perform static analysis, specialised tools are more commonly used due to their ability to automate the analysis process. This makes them more efficient and less prone to human error.

These dedicated static analysis tools are able to flag various issues, including syntax errors, coding standard violations, code complexity, unused code, null pointer dereferences, security vulnerabilities and even performance bottlenecks [8, 36, 50]. Static analysis is therefore useful for catching potential issues early in the development cycle, reducing the likelihood of bugs and security vulnerabilities, and improving code quality, maintainability and overall software reliability [2].

Static analysis also aids in improving the overall codebase by highlighting areas of code that may be overly complex or redundant. This enables code refactoring for better efficiency and maintainability. Furthermore, it is particularly valuable in the context of code reviews and collaboration among development teams, since it can serve as an additional inspection layer that complements the human review process [42]. This helps to ensure that coding standards are followed consistently.

2.2. Statically-Typed vs. Dynamically-Typed Languages

Statically-typed programming languages like Java and Scala, see Figures 2.1a and 2.1c, offer a robust mechanism for type safety that catches errors at compile-time rather than waiting until runtime. This proactive approach reduces the likelihood of encountering common programming mistakes that can lead to bugs and crashes [12, 33], thus enhancing the overall stability and reliability of the codebase. The benefits of type safety are further emphasised by a comprehensive large-scale user study that suggests that programmers indeed get substantial advantages from using statically-typed languages [16]. Moreover, the type information

```java
public static int addNumbers(int x, int y) {
    int addition = x + y;
    return addition;
}
```

(a) Java

```python
def add_numbers(x, y):
    addition = x + y
    return addition
```

(b) Python

```scala
def addNumbers(x: Int, y: Int): Int = {
  val addition = x + y
  addition
}
```

(c) Scala

```javascript
function addNumbers(x, y) {
  let addition = x + y;
  return addition;
}
```

(d) JavaScript

Figure 2.1: Examples of statically-typed programming languages (left) and dynamically-typed languages (right).

provided by statically-typed languages improves code readability and understanding, since explicitly declaring variable types in the source code serves as valuable documentation. This documentation helps both developers and maintainers understand the intended usage of the variables. Additionally, this type information empowers IDEs and other development tools to offer advanced features such as more precise code completion and comprehensive error checking [40]. These tools leverage the static nature of the language to provide real-time feedback and assistance, contributing to a more efficient and streamlined development process.

Conversely, dynamically-typed programming languages like Python and JavaScript, see Figures 2.1b and 2.1d, adopt a more flexible approach by determining variable types at runtime. A variable type is inferred based on the value assigned to the variable when the program runs [49] and thus these languages either lack or do not require type annotations. This allows developers to change the type of a variable at any point during the execution of the program [25]. This flexibility can simplify the code-writing process, as developers are not constrained by explicit type declarations. However, this dynamic typing approach also introduces challenges, particularly in debugging, as type errors may only surface during runtime. The absence of compile-time type checking can make it more challenging to catch and address issues early in the development process, potentially leading to more time-consuming debugging efforts.

2.3. Classical Type Inference

Classical type inference is a process to automatically deduce or infer the types of expressions and variables without requiring explicit type annotations from the programmer [48]. The compiler or type checker analyses the program's syntax and expressions to determine the types of variables and expressions at compile-time based on their usage and context. It uses a set of rules and algorithms to infer types and ensure type consistency throughout the program.

Examples of these rules include type deduction, type inference and type constraints [30, 44, 45]. With type deduction, the compiler analyses the program's structure, usage and context to deduce the types of variables and expressions. As for type inference, type information is propagated through the program, allowing the compiler to determine the types of variables and expressions based on their usage. Lastly, the inference process may involve generating and solving constraints that represent relationships between different types. These type constraints are then integrated to guide the type inference algorithm in determining the most general and consistent types for variables and expressions.

A simple example of classical type inference is shown in Figure 2.2 where the Scala compiler can infer the return type of the function as an integer.

```scala
def squareInt(num: Int) = {
    num * num
}

val square = squareInt(2)
println(square.getClass)  // prints int
```

Figure 2.2: Scala multiplication function with inferred integer return type.

2.4. Machine Learning Type Inference

Although classical type inference generally performs well, it does have its limitations. There are instances where the type inference algorithm may encounter ambiguous situations where it cannot determine the precise type of an expression. Furthermore, relying on a set of rules and constraints may hinder its ability to handle certain advanced language features or complex type systems effectively [5]. Therefore, using machine learning methods for type annotation inference can mitigate some of these limitations of classical type inference, enabling the determination of type annotations in places where that would otherwise not be possible.

In recent years, there has been a growing interest in applying machine learning methods to infer type annotations for dynamically-typed languages like Python or JavaScript. Several studies, such as [17, 22, 26, 28, 29, 31, 46, 47, 51], have explored these methods to accurately predict type annotations. These methods make use of function names, parameter names, code comments and other contextual information in the source code to train a model to predict type annotations. The resulting predictions typically consist of multiple type annotations along with associated probabilities, where higher probabilities indicate greater likelihood of correctness for the predicted type slot.

2.5. Call Graphs

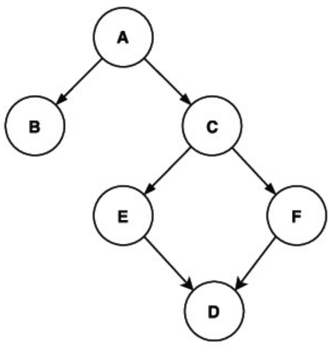
Call graphs model the relationships between different functions and methods in a program, showing how the control flows from one function to another [37, 43]. In this thesis, they serve as a means to assess the downstream impact on program analysis, as discussed in Research Question 3. They can be constructed statically or dynamically. Static call graphs are generated by analysing the source code or intermediate representations of a program without executing it. Dynamic call graphs, on the other hand, are constructed at runtime by monitoring the actual function calls during program execution. This research focuses specifically on static call graphs as those are related to static analysis. Therefore call graphs are assumed to be generated statically in further references.

Each node in a call graph represents a function or method and the edges between the nodes indicate the flow of control between these functions. Typically, the caller function is connected to the callee function by an edge, indicating that the caller function invokes or calls the callee function. Figure 2.3 shows how a program (a) can be represented as a call graph (b). A call graph is thus able to provide valuable insights into the structure and behaviour of a program, allowing developers to understand how different parts of the program interact with each other. On top of that, it can also be used to identify the dependencies between functions, even from imported packages.

```python
def caller():
    A()
    if condition1:
        B()
    else:
        callee()

def callee():
    C()
    if condition2:
        E()
    else:
        F()
    D()
```

(a)



(b)

Figure 2.3: Call graph (b) constructed from corresponding Python code (a). This figure was based on [10].

Although a call graph can provide these valuable insights, it also has a couple of challenges that it faces. First of all, on large-scale programs, call graphs can become extremely large and complex. As the program size increases, the number of functions and the complexity of their interactions grows exponentially. This makes it challenging to visualise and navigate the call graph effectively, leading to difficulties in identifying specific call paths. Furthermore, programs that rely heavily on frameworks or libraries are also more troublesome as call graphs may not capture the interactions with external code accurately. Lastly, constructing accurate call graphs for programs written in dynamic languages can be more challenging since these languages contain features like dynamic typing, late binding and runtime code modification. These features make it difficult to determine the exact function calls during static analysis [38]. To solve several of these aforementioned challenges, this thesis hypothesises that adding more type annotations in Python code provides additional information to the call graph generation tool which it can use to generate a more precise call graph.

3 Related Work

The chapter provides an overview of prior research efforts and various methods focused on enhancing type inference using learning algorithms. Additionally, it identifies a research gap and provides further elaboration on this.

3.1. Related Work

The integration of type annotations into Python started in 2014 with the proposal of PEP 484 [35] by the Python community, outlining the addition of optional type annotations to Python programs. This initiative was realised with the release of Python version 3.5, which introduced optional type annotations alongside the Mypy [20] type checker. Further advancements in the Python ecosystem following the introduction of PEP 484 led to the creation of other type checkers like Pyre [9], Pytype [14] and Pyright [24].

During this period, machine learning techniques also began gaining traction. They offered the potential for predicting type annotations that could be integrated into source code. The best-known machine learning models and their impact on predicting and integrating type annotations are discussed below.

JSNice The field of learning-based type inference started with an approach by Raychev et al. [34] in 2015. They learned a probabilistic model from existing data which could predict the properties of new and unseen programs. This approach was realised in JSNice, a scalable prediction engine utilising conditional random fields (CRFs) to predict identifier names and type annotations of variables for JavaScript.

PyProbaTyping Xu et al. [51] propose a Python type inference technique based on probabilistic inference. They observed that Python programs contain numerous type hints, including accessed attributes, variable names and explicit type checks, which are often uncertain. Their idea was to correlate all these uncertain type hints, which are propagated and aggregated among program artefacts (e.g., data flow between variables) in a probabilistic model to predict probabilities of variable types.

DeepTyper Thereafter, Hellendoorn et al. [17] introduced DeepTyper in response to the contextual limitations of JSNice. DeepTyper is a sequence-to-sequence neural network model trained on an aligned corpus of TypeScript code. Although it can use a much wider context to predict type annotations, its probabilistic nature leads to “type drift”. This occurs when the probabilities in the type vector of a variable change throughout its definition and usage in the code, despite the true type of the variable remaining fixed. This results in difficulties maintaining consistent predictions for the same variable.

NL2Type NL2Type [22] was the next proposed model that aimed to predict type annotations for JavaScript functions. Malik et al.'s key idea was to exploit natural language information in source code, such as comments, documentation, function names and parameter names to predict types. They train a recurrent, LSTM-based neural model that is shown to outperform both JSNice and DeepTyper.

TypeWriter Inspired by the NL2Type model for JavaScript, Pradel et al. [31] introduced TypeWriter, a deep neural network model designed to predict type annotations for Python. It incorporates both code context and natural language information from the source code to infer the return and argument types for functions. Additionally, TypeWriter employs a combinatorial search strategy to validate the predictions of its neural model by using feedback from an external type checker.

LambdaNet LambdaNet [47], created by Wei et al., proposes a probabilistic type inference scheme for TypeScript based on a graph neural network. A lightweight source code analysis is performed to create a type dependency graph, connecting type variables with logical constraints, name and usage information. Then, a pointer-network-like graph neural network is used to propagate information between related type variables. This enables type predictions for both standard types and user-defined types that were not even encountered during training.

OptTyper Natural type inference (NTI) uses natural language text within source code to determine valid type annotations. Although techniques based on NTI are empirically effective, they are not sound by construction. Pandi et al. [28] introduce the first algorithm for NTI validated with theorems and proofs. Their tool called OptTyper leverages both logical constraints (derived from type rules) and natural constraints (arising from the natural language text associated with a variable and its uses) to be used in a joint optimisation problem to determine types for TypeScript. The logical constraints are treated as hard constraints and the natural constraints as soft constraints. This results in a formal proof of soundness that demonstrates that the algorithm always terminates with either an error or a guaranteed type annotation for its input.

Typilus Most of the aforementioned learning-based type inference methods utilise a limited-size type vocabulary, for instance, 1,000 types. This limitation hinders their capability to deduce user-defined and rare types. To address this issue, Allamanis et al. [1] propose Typilus, a graph neural network model that predicts types in Python code by probabilistically reasoning over the program's identifiers, syntactic constraints and patterns, and semantic properties like control and data flow. By using one-shot learning, an open vocabulary of types can be predicted, including rare and user-defined ones. Furthermore, Typilus can also find incorrect type annotations.

TypeBert Commonly, type inference models use different (hand-engineered) inductive biases, ranging from simple token sequences to complex graphical neural networks (GNNs), in order to predict type annotations. Jesse et al. [18] challenge the need for sophisticated inductive biases, proposing the use of "big data" to learn natural typing patterns instead. Their TypeBert model, based on a pre-trained transformer model, demonstrates that type annotation performance of the most sophisticated models can be surpassed with a simple token-sequence inductive bias used in BERT-style models and enough data.

Type4Py Mir, et al. [26] introduce Type4Py, a hierarchical neural network model that uses deep similarity learning. This model discriminates between similar and dissimilar type annotations in a high-dimensional space, forming clusters of types. The model then infers likely type annotations for arguments, variables and return values using a nearest neighbour search within these clusters.

HiTyper Next, a hybrid type inference approach named HiTyper [29] is presented by Peng, et al. Their main idea is that static and DL-based approaches offer complementary benefits and thus a novel approach is introduced that integrates both rule-based static inference and deep learning for precise type annotations. Their key innovation is the use of Type Dependency Graphs (TDGs) to record type dependencies among variables within functions. TDGs allow the integration of type inference rules and type rejection rules. This enables static inference and correctness inspection of deep learning-based predictions until the TDG is fully inferred. For the deep learning predictions, both the Typilus and Type4Py models are used.

DiverseTyper Although Jesse, et al. showed superior performance in predicting common type annotations with their TypeBert model, they did notice that the performance on user-defined types could still be improved upon. So later, they introduced DiverseTyper [19], an improved TypeBert model using deep similarity learning, in order to predict user-defined types more accurately.

TypeT5 Wei, et al. [46] introduce TypeT5, a type inference method that utilises CodeT5, a pre-trained language model for code. TypeT5 treats type prediction as a code infilling task. Using static analysis, dynamic contexts for each code element are build. For these, the type signature can then be predicted by the model. Furthermore, it also uses an iterative decoding scheme that integrates previous type predictions into the model's input context. This allows for information exchange between related code elements. Not only does this lead to higher overall accuracy, especially on rare and complex types, but it also produces more coherent results with fewer type errors.

OpenTau With the rise of Large Language Models (LLMs) and their general capabilities that are constantly evolving, it can be wondered whether they can be applied to the task of performing type inference. Challenges such as poor performance in fill-in-the-middle tasks, context window size limitations, potential type-checking issues in generated types, and the difficulty in measuring the type quality of the output program are valid concerns related to LLM type predictions. To address these challenges, Cassano, et al. [6] created OpenTau, a search-based type prediction approach that utilises LLMs. Their contributions consist of a fill-in-the-type fine-tuning method for LLMs, a new metric for assessing type prediction quality and a tree-based program decomposition method for exploring generated types.

CodeTIDAL5 Recent approaches of statistical techniques to predict type annotations based on machine learning show overall improved accuracy. However, they still perform significantly worse on user-defined types than on the most common built-in types. Furthermore, they rarely integrate with user-facing applications, thus limiting their real-world usefulness even more. Because of these constraints, Seidel, et al. [39] developed CodeTIDAL5, a transformer-based neural type inference model based on CodeT5. This model uses source code context and extracted usage slices from a program's code property graph to query variable types in JavaScript/TypeScript. CodeTIDAL5 can predict type annotations and provide hints on unseen types, thus improving effectiveness on user-defined types. Moreover, the authors present JoernTI, an integration of their approach into Joern, an open-source static analysis tool. This availability through Joern improves real-world usefulness as developers have easier access to it.

3.2. Research Gap

Most of the aforementioned type inference models predict type annotations on user-provided code, but cannot guarantee the correctness of these predicted types. For instance, a model may predict three type annotations for a type slot, where only the second prediction is valid within the context of the codebase. Or even none of the predictions are allowed as they would result in a syntax error. While improving prediction accuracy helps to mitigate the correctness issue to some extent, no model achieves 100% accuracy. Therefore, incorporating methods to verify the correctness of predicted type annotations can lead to improved practical utility.

TypeWriter [31] already uses this idea of using static type inference for checking the predictions of the machine learning model. However, its machine learning model is outperformed by other works, its search-based validation cannot correct the wrong types but only filter them out and the main constraint is the time that it takes to get feedback from the type checker. OptTyper [28] then improves upon this by incorporating both logical and natural constraints into a single prediction step. The logical constraints are created by “relying on a mode of operation where the compiler infers some types from usage on TypeScript code”. However, the authors do recognise that less information is obtained this way than if they had used a full type checker. HiTyper [29] also combines both static and DL-based approaches by using a Type Dependency Graph. The TDG is filled iteratively with predictions from both static type inference and deep learning-based predictions such that no type inference rules are violated. This is an alternative approach to the one presented in this thesis.

Ultimately, the identified research gap specifically lies in the need for a fast search strategy capable of validating combinations of predicted type annotations for Python code. This ensures their validity within the context of the codebase and increases the total number of type annotations.

4 Approach

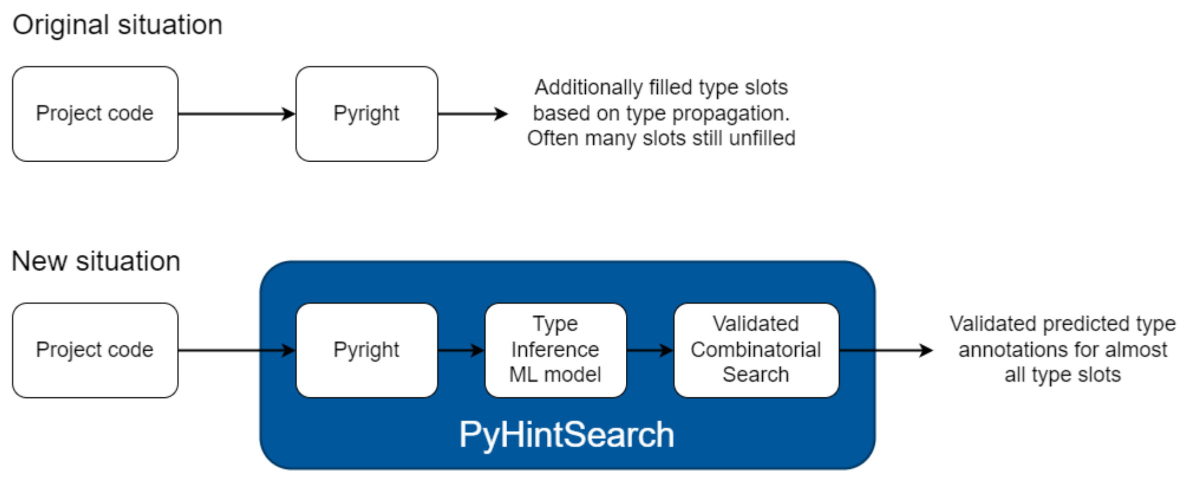

Original situation
Project code
Pyright
Additionally filled type slots based on type propagation. Often many slots still unfilled
New situation
Project code
Pyright
Type Inference ML model
Validated Combinatorial Search
PyHintSearch
Validated predicted type annotations for almost all type slots

Figure 4.1: Situation overview of Pyright vs. PyHintSearch. Currently, Pyright can fill some type slots based on type propagation, but leaves many unfilled. PyHintSearch can fill the remaining type slots with a searched combination of ML-predicted type annotations that fits in the code context.

Any Python project compatible with version 3.5 or above can be type annotated. Since the annotations are optional, developers have the flexibility to decide whether to integrate them or not. It is even possible to gradually add more types to a project over time, however, due to the labour-intensive nature of manual type annotation, many projects either lack type annotations entirely or are only partially annotated. The major advantage of incorporating type annotations is their utility for static analysis by a static type checker. This type checker assesses the validity of a program and, if the program passes the check, it is guaranteed to satisfy some set of type safety properties. Furthermore, such a type checker can often also infer several type annotations by statically analysing the code, which can then be added to the codebase. However, this process often leaves many type slots unfilled. Utilising machine learning predictions and a validated combinatorial search in the developed tool called PyHintSearch, has the potential to decrease the number of unfilled type slots. Figure 4.1 provides an overview contrasting the original situation using only a type checker, such as Pyright, with the new situation using PyHintSearch.

4.1. Pyright Static Type Checker

According to the Pyright website [24]: “Pyright is a full-featured, standards-based static type checker for Python. It is designed for high performance and can be used with large Python source bases.” This tool, developed by Microsoft, is primarily used for checking whether variables and functions are used correctly in the code, although it can also be used to infer type annotations based on a set of rules. Moreover, Pyright includes a language server, which can be manipulated in order to provide feedback on the validity of newly added type annotations. This is especially important during the combinatorial search which tries to identify a valid combination of type annotations.

As mentioned, Pyright has the ability to infer type annotations. This can be achieved by opening a terminal and running the `pyright --createstub` command in order to generate Python stub files. Python stub files, often referred to as type hint stubs, are files containing type hints without the actual implementation code. They use the “.pyi” file extension instead of “.py”. These stub files are used to provide static type information to tools like static type checkers, IDEs and other tools that analyze Python code without executing it. This command is intended to give a “first draft” of a type stub, which the user then needs to manually edit to make it into a workable stub. Although this does sound labour intensive, most of the time the generated type stub can be used directly.

The main advantage of letting Pyright generate these stub files, is its ability to make a good guess for the return type of a function. However, since it is only a guess, it gets added as a comment in the stub file. Note that Pyright is actually rather accurate with its inferred types. By parsing the return type comments, these annotations can be integrated in the original code resulting in more type slots being filled in already.

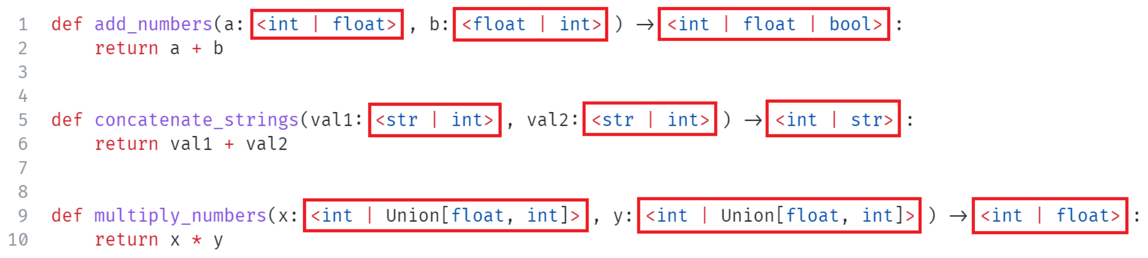
Although Pyright is able to infer type annotations, it can usually not determine all available type slots, meaning that there are still empty type slots left. Static analysis works best when Python code has as many annotations as possible, so therefore another method needs to be used to fill in the remaining type slots.

4.2. Optimized Combinatorial Search

While static type inference techniques are precise, they frequently leave type slots unfilled. Machine learning approaches, on the other hand, can predict type annotations for each slot, yet they cannot guarantee the correctness of the predicted types. Therefore, a search strategy is needed to find a combination of predicted type annotations that is valid within the context of the codebase.

4.2.1. Machine Learning Top-*n* Predictions

Commonly, a machine learning model predicts multiple type annotations for each type slot which are then ranked based on their probability. The type annotation with the highest probability is the most likely one to fit in that type slot, followed by the second prediction, third, etc. Figure 4.2 presents several basic Python functions, for which each type slot, marked in red, has multiple predicted type annotations ranked on their probability from left to right. For example, for parameter `a` of the `add_numbers` function, the most likely type annotation is `int`, followed by `float`.



```
def add_numbers(a: <int | float> , b: <float | int> ) → <int | float | bool> :
    return a + b


def concatenate_strings(val1: <str | int> , val2: <str | int> ) → <int | str> :
    return val1 + val2


def multiply_numbers(x: <int | Union[float, int]> , y: <int | Union[float, int]> ) → <int | float> :
    return x * y
```

Figure 4.2: Multiple predicted type annotations for the marked-in-red type slots. Identifying a valid combination of these for is the goal of the combinatorial search.

If only the top-1 prediction for each type slot is taken, the `add_numbers` function would be annotated as follows:

```python
def add_numbers(a: int, b: float) -> int
```

This is an invalid combination of type annotations, because the return type should be the second prediction, `float`, instead of `int`. Similarly, the `concatenate_strings` function would be incorrectly annotated as:

```python
def concatenate_strings(val1: str, val2: str) -> int
```

The top-3 or even top-5 predictions can also be used when trying to determine a combination of valid annotations for the presented functions. This, however, is beginning to show how finding such a combination of type annotations becomes increasingly more difficult because of the exponentially growing search space. Let's take, for example, the top-3 predicted annotations to be used. For each type slot, up to 3 annotations could be checked when none of the predicted types fit. Therefore, for n type slots, there are 3^n potential combinations and for the top-5, this would be 5^n. This combinatorial explosion should be avoided to keep annotating functions feasible.

4.2.2. Search Process and Backtracking

Using a **greedy approach** is a strategy to address this explosion of combinations. Initially, the type annotation with the highest probability is applied to the type slot. To check the correctness of this type annotation in the context of the code, the feedback from a static type checker, such as Pyright in our case, is used. If Pyright deems the type annotation valid, the process continues to the next type slot. Otherwise, the next predicted annotation is attempted. This process repeats until a valid annotation is identified, or none of the predicted annotations prove valid, in which case an empty annotation is inserted. An empty annotation indicates that the slot remains unfilled, thus satisfying Pyright's criteria for validity, allowing for the exploration of the next type slot.

One thing that makes greedily filling in the type slots more complicated is the interdependencies among functions. Consider two functions, `foo` and `bar` for which several type annotations are predicted by a machine learning model. The `bar` function has its return type slot already annotated with an `int` annotation, but all other slots are yet to be filled. The implementation and predicted type annotations of both `foo` and `bar` functions are illustrated in Figure 4.3.

```python
def foo(value: <str | int>) -> <int | float | List[bool]>:
    multiple_of_3 = 3 * value
    return multiple_of_3

def bar(x: <int | bool | List[float]>) -> int:
    number = foo(x)
    return 42 + number
```

Figure 4.3: Implementation and predicted type annotations for `foo` and `bar` functions.

First, the predicted `str` annotation is tested for the parameter of `foo`. This is valid according to Pyright, thus the process continues to the return type. Here the `int`, `float` and `List[bool]` annotations are tried, all of which are invalid. However, filling in the empty annotation, i.e. nothing, is accepted. Then, the parameter `x` of the `bar` function is tried.

Neither `int`, `bool`, `List[float]`, nor the empty type annotation are valid due to the dependency on `foo` in the function body, which expects a `str` parameter and implicitly returns a `str` value. Since none of the tried predictions are valid, backtracking to the previous type slot is necessary. In this case, the return value of the `foo` function is visited again. But, since all the predicted annotations have been tried there already, the need for backtracking rises once more. This time, it ends at the type slot of `foo`'s parameter. The next predicted annotation, `int`, is now attempted. This is valid, therefore continuing the process to the next type slot, which is again the return type. All type annotation predictions are retried for this slot, starting with the first value, `int`. This annotation succeeds, after which the `bar` function is visited. Once again the predicted types are tried, starting with `int`, which is now valid. Consequently, all type slots are filled, and a valid combination of predicted type annotations is identified. This example shows that backtracking was essential for exploring the search space for a valid combination and an overview of this process is presented in Figure 4.4.

```python
def foo(value: str):                # Correct
# Continue to next slot
def foo(value: str) -> int:         # Incorrect
def foo(value: str) -> float:       # Incorrect
def foo(value: str) -> List[bool]:  # Incorrect
def foo(value: str):                # Correct
# Continue to next slot
def bar(x: int) -> int:             # Incorrect
def bar(x: bool) -> int:            # Incorrect
def bar(x: List[float]) -> int:     # Incorrect
def bar(x) -> int:                  # Incorrect
# Need to backtrack (2x)
def foo(value: int):                # Correct
def foo(value: int) -> int:         # Correct
def bar(x: int) -> int:             # Correct
```

Figure 4.4: Backtracking during the combinatorial search for the interdependent `foo` and `bar` functions.

4.2.3. Datastructure

Given the interdependence among functions and the need for backtracking to explore new combinations, it may be wondered which data structure encapsulates this problem well and can be used to efficiently search the numerous potential combinations. The answer to this is a **search tree** with branches related to the top-n predictions that are searched. In constructing the search tree, the first n type annotations predicted by the machine learning model are taken into account. These annotations, ranked by probability, form the branches of the tree from left to right. Additionally, an extra branch on the right indicates an empty type. Figure 4.5 illustrates a sample search tree featuring the top-3 predicted type annotations, which are systematically explored during the search process.

The search tree starts from the top node, representing the first available type slot. Initially, the `int` type annotation is attempted in the slot. If deemed valid, progression along the branch continues to the next node, corresponding to the next type slot. In the event of the `int` type annotation being invalid, the `bool` type annotation is considered next. Each time a valid type annotation is found, progress continues to the next layer of the search tree, thus creating a branch of valid types. If every layer that represents an available type slot has been explored, and a valid type annotation has been identified for each type slot, then the branch that contains a valid combination of type annotations for all type slots is discovered. This means that the combinatorial search has successfully found additional type annotations for a file. Note that there is a possibility that a path along a branch is stuck at a certain type slot with none of the predicted type annotations being valid. In that case, a backtracking

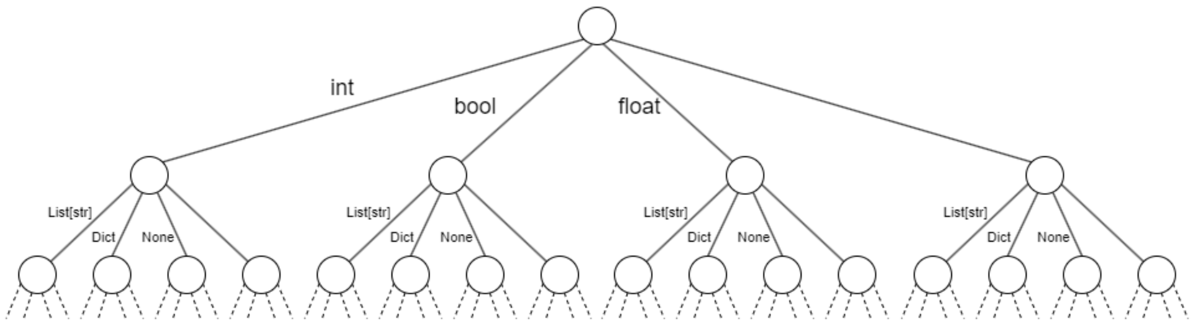


Figure 4.5: Top-3 predictions search tree. The highest-rated prediction is the leftmost branch of a node, followed by the other predictions in order of probability. The right-most branch is the empty type, which leaves the type slot empty. Each layer of nodes corresponds to a type slot to be filled.

approach is used by going up one layer and trying the next type annotation, similar to a **depth-first search**.

Ultimately, the goal is to discover a valid combination of machine learning type annotations, along with those determined by Pyright, to incorporate into the Python code. This combination results in an increased number of filled type slots, addressing the limitations where Pyright alone might fall short. Together, these methods complement each other, leading to enhanced type annotation coverage of Python code, thus contributing to improved static analysis.

4.3. Components of PyHintSearch

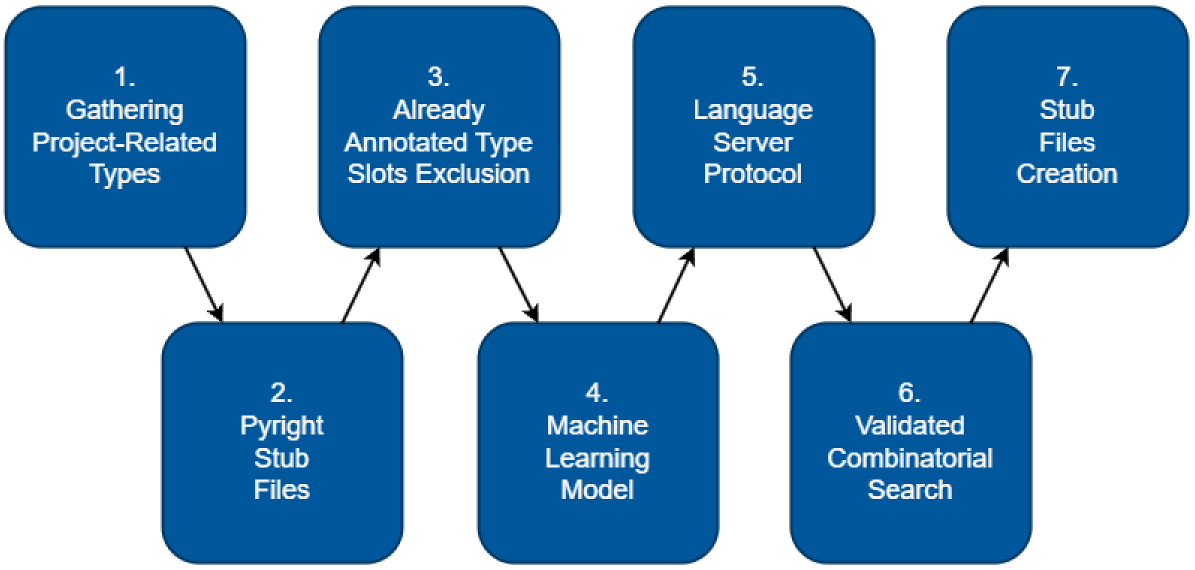

1. Gathering Project-Related Types
2. Pyright Stub Files
3. Already Annotated Type Slots Exclusion
4. Machine Learning Model
5. Language Server Protocol
6. Validated Combinatorial Search
7. Stub Files Creation

Figure 4.6: Components of the PyHintSearch tool.

The combinatorial search strategy outlined in Section 4.2 is combined with various practical components in a tool called PyHintSearch. See Figure 4.6 for an overview of its components. PyHintSearch's source code can be found at: `https://github.com/FrostMegaByte/py-hint-search`

4.3.1. Gathering Project-Related Types

The PyHintSearch tool that this thesis introduces, works on a user-provided project that consists of Python files that need to be type annotated. Typically, such projects include classes that can be used as type annotations. Additionally, Python projects often have dependencies, which can be installed in a virtual environment. These dependencies also contain classes that can be used as type annotations. Therefore, it is necessary to gather all local project classes and their corresponding file locations to ensure that a class can be imported when it is predicted as a type annotation. If a virtual environment is present, the user can specify its location as a parameter in the command line to allow for the collection of all classes from the project dependencies including their file location. Similarly to local classes, this information is needed to import the class when it is predicted as a type annotation.

4.3.2. Pyright Stub Files

As discussed in Section 4.1, Pyright is a static type checker for Python that is also capable of inferring type annotations with the `pyright --createstub` command. The output of this command is a “first draft” of a type stub, containing inferred type annotations for the return type of the defined functions. The `pyright --createstub` command only seems to work properly on packages that can be imported and not on individual Python files. Additionally, if there is a directory containing Python files with inside that directory other nested subdirectories containing Python files, then the nested subdirectories are skipped when generating the type stubs. The solution to both of these problems is as follows: 1) Traverse the directory structure from the bottom up. So start in the most deeply nested directories and work upwards to the specified directory location. And 2) create a temporary `__init__.py` file in each directory, such that type stubs are generated for all Python files in that directory. Both of these steps have been implemented in the `pyright_typestubs_creator.py` file which allows a user to specify the directory location of Python files for which Pyright should create type stubs that contain inferred return types.

4.3.3. Exclusion of Already Annotated Type Slots

For each type slot, the machine learning model determines one or multiple type annotations to try during the validated combinatorial search. However, as mentioned before, this search can grow exponentially based on the number of type slots that are tried. Therefore, speeding up the search by reducing the number of slots to consider is crucial for efficiency. To achieve this, all type annotations that are initially present or can be determined by other means are excluded from the search.

The LibCST[1] Python package assists in the exclusion of already annotated type slots. This tool parses Python code into a concrete syntax tree (CST), enabling the extraction of type annotations for each type slot. This allows for the identification of type slots that already contain annotations and slots which do not. Additionally, LibCST can also be applied to the generated Pyright stub files, creating CSTs from which additional type annotations can be extracted. These annotations are then integrated into the program CST, enhancing the number of type annotations and resulting in extra type slots that can be excluded from consideration. Consequently, this refinement of the search space enables the combinatorial search to concentrate exclusively on the remaining unfilled type slots.

4.3.4. Machine Learning Model

Several machine learning models, such as Typilus [1], Type4Py [26], HiTyper [29], TypeT5 [46], LambdaNet [47], and others, have been developed to predict type annotations. These models try to predict one or multiple type annotations for a type slot.

The PyHintSearch tool is designed to be agnostic to the specific machine learning model used for type annotation prediction. As long as a model can generate type annotation predictions for type slots in the user-provided code and output its predictions in a specific format, it is suitable for use with PyHintSearch. The specific format that PyHintSearch uses is Type4Py’s JSON response convention[2]. This is chosen as Type4Py is the default model used by PyHintSearch and its response works well as a web API. Therefore, as long as a model outputs

[1]`https://github.com/Instagram/LibCST`
[2]`https://github.com/saltudelft/type4py/wiki/Using-Type4Py-Rest-API#json-response`

its predictions in this format, PyHintSearch can parse the JSON and use the predictions to construct a search tree that is used for the combinatorial search.

Although any machine learning type inference model that conforms to the aforementioned JSON response format can be integrated with PyHintSearch, this thesis primarily focuses on using Type4Py and TypeT5. Type4Py, a model developed by the Software Engineering Research Group at Delft University of Technology, has a public API endpoint (`https://type4py.com/api`) to which a file can be uploaded. After analysing the file, it responds with a JSON object containing the predictions of all type slots. Additionally, the model can be run as a local Docker container with the same functionality as the public API. This model was chosen because of its ease of use, making it well-suited for prototyping the tool, the availability and accessibility of the authors at Delft University of Technology and its accuracy, although surpassed by newer models, still being rather high.

On the other hand, the TypeT5 model was chosen as it was state-of-the-art at predicting type annotations when starting the implementation of PyHintSearch. The TypeT5 GitHub repository[3] contains clear installation instructions to get the model up and running, making it suitable for prototyping and testing its integration into PyHintSearch. While getting the model to produce predictions is not the challenging part, obtaining predictions for a user-provided project and making them available in the required JSON format via a web API is. To address this challenge, TypeT5-API[4] was developed. This Flask-based web API converts predictions from the TypeT5 model into the necessary JSON format required for integration with PyHintSearch. Since TypeT5 can predict type annotations for function parameters, return types and variables, whereas PyHintSearch only requires types for parameters and the return type, the predictions for variables have been disabled to speed up the inference process. A Dockerfile was created to simplify the deployment of TypeT5-API, allowing users to easily obtain predictions by providing the project path as a volume parameter when running the Docker container. As a result, the TypeT5 model is fetched and applied to the project, generating predictions in the required JSON structure, which are then sent to the `http://localhost:5000/api` endpoint.

4.3.5. Language Server Protocol

Static type checkers like Pyright use type annotations to detect type-related errors such as type mismatches or incorrect usage of functions or methods. Besides, they can also be exploited to give feedback on the correctness of inserted type annotations. In this thesis, Pyright serves as an oracle to determine whether inserting a type annotation is allowed or would cause an error, prompting the next type annotation to be considered. Given the need to evaluate a lot of combinations, it is important that feedback from Pyright is received quickly. Ideally, this means that Pyright's built-in functions can be used to analyse an inserted type annotation to get diagnostics fast.

Pyright itself is implemented in TypeScript, making direct access to its internal functions and data structures inaccessible from Python. However, Pyright also features a language server that is core to the Pylance[5] plugin in VS Code. This language server can be used to analyse source code files. In VS Code, for example, users can enable the option to let Pylance perform a strict analysis on opened files to determine any static analysis errors in the code. This aligns perfectly with the desired functionality of verifying the validity of added type annotations. For example, a type annotation is inserted in the code and the static analysis indicates that it is not allowed in that context. This feedback is then used to guide the combinatorial search.

To exploit Pyright and enable this functionality, the approach in this thesis involves faking being a code editor using the Language Server Protocol (LSP) [23]. The LSP uses the JSON-RPC protocol to send messages related to specific actions between development tools and the

[3]`https://github.com/utopia-group/TypeT5`
[4]`https://github.com/FrostMegaByte/TypeT5-api`
[5]`https://marketplace.visualstudio.com/items?itemName=ms-python.vscode-pylance`

language server. It is predominantly used to power language features in a code editor like auto-completion, go-to-definition, find-all-references, syntax highlighting and more. Beyond these functionalities, it also defines which document is currently opened, what changes are being made to it and when it is closed. Interestingly, the Pyright language server recomputes its analysis upon detecting changes in the contents of a file. This feature is useful, as it reanalyses a file after adding an annotation in a type slot, producing diagnostics that are used to determine the validity of the annotation. The approach to achieve this in practice involves using the LSP to send commands to Pyright's language server, effectively simulating actions such as file opening, content modification and file closure. Hence, Pyright's analysis is performed, returning diagnostics that provide valuable feedback for the combinatorial search.

To start the communication between PyHintSearch and Pyright's language server, the language server must first be started. This is accomplished by executing the command `pyright-langserver --stdio`, enabling communication via the standard input and output. All further communication involves JSON-RPC messages transmitted over the standard input and output, containing instructions for both the language server and PyHintSearch.

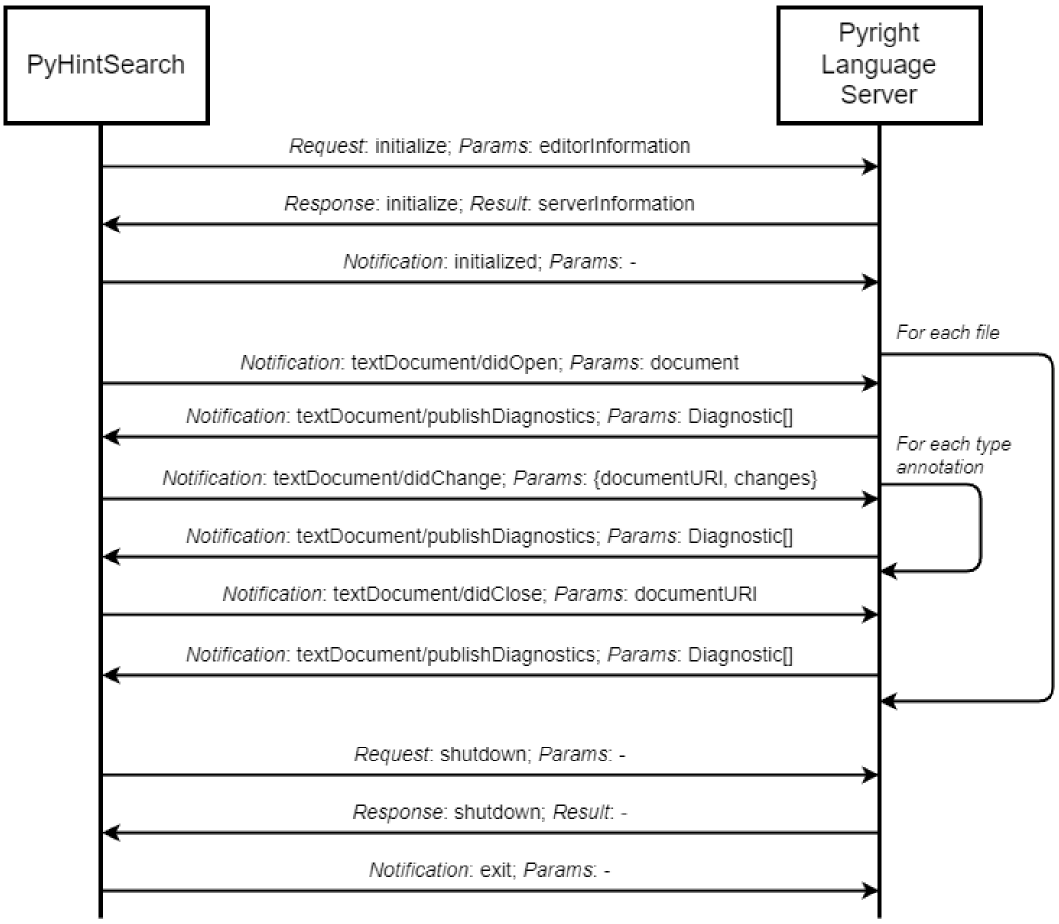

PyHintSearch
Pyright Language Server
Request: initialize; Params: editorInformation
Response: initialize; Result: serverInformation
Notification: initialized; Params: -
For each file
Notification: textDocument/didOpen; Params: document
Notification: textDocument/publishDiagnostics; Params: Diagnostic[]
For each type annotation
Notification: textDocument/didChange; Params: {documentURI, changes}
Notification: textDocument/publishDiagnostics; Params: Diagnostic[]
Notification: textDocument/didClose; Params: documentURI
Notification: textDocument/publishDiagnostics; Params: Diagnostic[]
Request: shutdown; Params: -
Response: shutdown; Result: -
Notification: exit; Params: -

Figure 4.7: LSP communication between PyHintSearch and Pyright's language server.

Initially, an `initialize` message is sent, containing information related to the capabilities of PyHintSearch's fake editor. The language server then responds with information about the server, followed by an `initialized` message back from the editor. Once the connection has been successfully established, specific actions can be communicated. For PyHintSearch, the fake editor is designed to be incredibly basic, implementing only file opening, changing and closing actions. For this purpose, the `fake_editor.py` file implements functions which correspond to sending `textDocument/didOpen`, `textDocument/didChange` and `textDocument/didClose` notifications.

The `textDocument/didOpen` message specifies which file should be opened, along with the version of the file. Upon receiving this notification, the server processes it and responds with initial diagnostics related to the file. Next, the *textDocument/didChange* message specifies all changes made to a specific file and increments the file version. PyHintSearch sends a message indicating that all the content in the file has changed, although typically only a single type annotation changes. While it is possible to specify partial content changes via the LSP, difficulties were encountered in implementing this. These difficulties relate to the complexity of specifying the removal of a type annotation. Furthermore, it remains uncertain whether sending partial changes accelerates the static analysis of a file. If it does, this enhancement could speed up the validated combinatorial search. Nonetheless, sending changes relating to all content in a file currently functions effectively, with the server responding with diagnostics indicating any warnings and errors in the file. These diagnostics are then used to validate the added or modified type annotation in a file. This feedback emphasises the utility of Pyright as a tool for validating type annotations, ensuring that no invalid annotations are introduced to the Python code. Once the search for a valid combination of type annotations for a file is complete, the `textDocument/didClose` notification is sent, indicating that the file should be closed. The language server then follows up with the final diagnostics related to the analysed file.

Once all the Python files are searched by PyHintSearch, two final LSP messages are sent. The `shutdown` request asks the server to shut down, but to not exit (otherwise the response might not be delivered correctly to the editor). Finally, the `exit` notification instructs the server to exit its process, completing the LSP communication. A complete overview of the communication between PyHintSearch and Pyright's language server is depicted in Figure 4.7.

4.3.6. Validated Combinatorial Search

The core focus of this thesis lies in the validated combinatorial search, outlined in Section 4.2. This section describes the implementation of the required search tree and the greedy search approach.

Once the machine learning model returns its predictions as a JSON response, the data must be structured into a search tree. First, the response is parsed to create a dictionary of all the type slots in the code and their associated predictions. Next, another dictionary is created to represent the actual search tree. Here, each type slot corresponds to a layer in the tree, with each key representing a layer and its associated value being another dictionary containing information about the type slot and its predictions. This approach minimises information duplication, as redundant data would otherwise be replicated for every node in the layer. To illustrate this, look back at Figure 4.5 and notice the three layers of nodes. The second layer contains four nodes with identical information. By storing this data once, rather than duplicating it for each node, the memory usage is optimised, preventing memory constraints caused by the exponential growth of the tree as the number of type slots increases.

The greedy search approach uses a depth-first search (DFS) method to explore the search tree efficiently. Unlike traditional DFS methods that use a stack to add and pop nodes for traversal, the tree's efficient storage in dictionaries requires a different approach. Instead, a two-pointer system is used. One pointer corresponds to the current layer, i.e. the current type slot, while the other pointer is in a list of indices that indicate the current prediction for each layer. This system is used to track the progress throughout the search tree, enabling DFS functionality without the memory overhead of storing the entire tree.

The search strategy processes the tree from left to right, inserting a type annotation into the original code that corresponds to an edge in the search tree. Next, the modified code is sent to the Pyright language server, which generates diagnostics related to the code. If the diagnostics confirm the validity of the added type annotation, the layer pointer index is incremented, advancing to the next layer. Conversely, if the diagnostics indicate an invalid type annotation, the index associated with the predictions for that layer is increased, attempting the next prediction for the type slot. If the index surpasses the number of top-n predictions, including the empty type, it indicates that no valid type annotation was found, thus the layer index gets decremented to facilitate backtracking. Ultimately, this search should identify a valid combination of type annotations given sufficient time. Two examples illustrating the successful role of backtracking in the search process are presented in Appendix A. Given that backtracking enables the exploration of all potential branches, leading to the examination of an exponential number of combinations, a 5-minute timeout was implemented for each file to prevent extremely long waiting times.

4.3.7. Stub Files Creation

Once a valid combination of type annotations is found, the results must saved. Throughout the combinatorial search, the LibCST concrete syntax tree of the program is continuously modified with type annotations, which is convenient, because it can eventually be saved as a fully annotated source code file. However, modifying the original code is not always desired, especially if it is someone else's library, for example. Therefore, saving the results as a stub file is preferable. With a few modifications, the annotated CST can be converted into a final stub file containing type-hinting information for the originally provided code.

The Mypy [20] static type checker includes the stubgen tool that can automatically generate stub files. Although stubgen was initially incorporated into the PyHintSearch tool, it had issues with generating stubs for deeply nested folders. This resulted in several projects missing their final type stubs and was therefore replaced with custom code that transforms a CST into a type stub file.

4.4. Evaluation Strategy

To answer the proposed research questions, the PyHintSearch tool needs to be evaluated on its effectiveness and efficiency. This section discusses four ways to evaluate the tool's coverage, correctness, performance and practical usage.

4.4.1. Preprocessing

Prior to evaluating the PyHintSearch tool on a project, the following steps are taken either beforehand or during the execution of the tool:

1. Python has a type stubs repository called "typeshed" [32], containing stub files for several Python projects. If stub files are available in typeshed or anywhere else for an evaluated project, their annotations are merged into the original user-provided project, creating a "fully annotated" version. Subsequently, PyHintSearch was applied to this version.

2. The Type4Py model is unable to parse Python code that contains the pipe symbol, i.e. `a | b` for union types in the newer syntax. Therefore, these symbols are transformed into `Union[a, b]` annotations.

3. In cases where `Incomplete` type annotations exist in the code prior to the machine learning model's prediction of type annotations, these are removed. This action opens up a type slot that can be filled with a predicted annotation instead.

4.4.2. Coverage Evaluation

In a simplified overview, the PyHintSearch tool consists of three steps. First, the initial step, in which a user provides a project, which may or may not contain existing type annotations in its source code. Second, the Pyright step, during which Pyright infers type annotations, which are then integrated into the code, serving as the baseline for the evaluation. Third, the validated combinatorial search step, during which a machine learning model performs type inference, predicting annotations for any remaining type slots, which are subsequently validated by a combinatorial search strategy. The results from this step are compared to the Pyright baseline to assess changes in coverage and determine whether machine learning complements static type inference. Throughout this process, various metrics are collected to evaluate the new outcomes against the baseline.

For each file of the user-provided project, the following metrics are collected:

- Filename
- Number of initial annotations
- Number of annotations after Pyright
- Number of annotations after PyHintSearch
- Number of fillable type slots
- Number of unfilled type slots
- Number of total type slots
- Number of extra Pyright annotations
- Number of extra PyHintSearch annotations
- Percentage of extra Pyright annotations
- Percentage of extra PyHintSearch annotations
- Percentage of all extra annotations
- Number of PyHintSearch evaluated type slots
- Percentage of extra PyHintSearch annotations after Pyright
- Number of arguments and return types, categorised into ubiquitous, common and rare groups, for the initial, after Pyright and after PyHintSearch type annotations. Annotations for Python's dunder methods are excluded, as these are considered too easy to determine.

While most of these metrics are fairly straightforward, understanding the "ubiquitous, common and rare annotations" metric requires further elaboration. For example, a question might arise regarding the categorisation technique. Since Python allows the same type to be written in different syntactic forms[6], there needs to be a method that can normalise and categorise annotations into their respective groups. The TypeT5 paper [46] describes a normalisation step that is performed to achieve this categorisation: "We recursively apply the following steps to normalise a Python type:

1. Rewrite any `Optional[T]` to `Union[T,None]`.
2. Sort the arguments of `Union` types and flatten any nested `Union`s.
   e.g., rewrite `Union[B,Union[C,A]]` into `Union[A,B,C]`.
3. If all type arguments are `Any`, drop them all. e.g., rewrite `List[Any]` to `List`.
4. Capitalise the names of basic types. e.g., rewrite `list` to `List`."

Furthermore, according to the LambdaNet paper [47], the top 100 annotations should cover 98% of all type annotations typically found in a project. Fortunately, the code for retrieving the TypeT5 model also obtains a list of the top 100 most frequent type annotations. The only change made to this list for the PyHintSearch tool involves replacing the `_MakeClient` annotation with the more commonly encountered `LiteralString` annotation. This decision was based on debugging the ubiquitous, common and rare groups and finding that `LiteralString` was used more frequently than `_MakeClient`, yet consistently being classified as rare.

Additionally, the Type4Py paper [26] further subdivides the common group by selecting the top 10 most frequent type annotations and calling these "ubiquitous". These annotations

[6]e.g., both `Union[int,None]` and `Optional[int]` refer to an integer that can also be `None`, and both `list` and `List[Any]` refer to a Python list with untyped elements

include `str`, `int`, `bool`, `float`, `List`, `Dict`, etc. For the evaluation of PyHintSearch, a similar approach is adopted, albeit with a slight modification. Instead of considering only the top 10 most frequent annotations, the list is expanded to include `None` and `Any` annotations as well. This adaptation is necessary because the correctness metrics discussed in Section 4.4.3 filter out `None` and `Any` types, thereby resulting in a revised list of the top 10 ubiquitous annotations matching the categorisation approach described by Type4Py.

4.4.3. Correctness Evaluation

The PyHintSearch tool merges the type annotations from Pyright with the identified valid combination of predicted type annotations, forming type stubs. While these determined type annotations are confirmed as valid through Pyright's check, it's important to note that their correctness is not guaranteed. For example, it can be the case that an `int` type annotation is added, whereas a`float` type was expected. Because `int` is a subset of `float`, it passes the validation, yet it might not fit in the original context. With this discrepancy in mind, a way to evaluate the correctness of type annotations is needed.

PyHintSearch takes already type annotated slots into account, such that it does not try to find new type annotations for them. This decreases the number of combinations that are searched, but also means that the predicted slots lack a corresponding "groundtruth" label in the original project code. Because of this, only the type annotations in the original project code can be used for the comparison against the new annotations. To ensure a fair comparison, all annotations from the original project code must be stripped before running the PyHintSearch tool. This now fills in the type slots that used to have a type annotation. After successfully running on the stripped project, the predicted types can then be compared to the original types to check their correctness.

To compare the correctness of the predicted type with the original type, a metric is needed to determine the quality and accuracy between the two. The TypeT5 paper [46] mentions three metrics for assessing type annotation accuracy, namely full accuracy, adjusted accuracy and base accuracy. Given that full accuracy does not remove `Any` and `None` type annotations, while the other two metrics do, this thesis opts to exclude full accuracy from the evaluation. **Adjusted accuracy** is used for a more meaningful comparison of correctness with prior works. "This metric (1) filters out all None and Any labels, (2) converts fully qualified names to simple names (e.g., `Tensor` instead of `torch.Tensor`), and (3) rewrites any outermost `Optional[T]` and `Final[T]` into `T` since they tend not to be used consistently across programmers." The **base accuracy** metric is the same as adjusted accuracy, except that it only checks the outermost type (e.g., `List[int]` will match any `List`, but not `Sequence`.)

Now that the number of added annotations matching the original ones can be determined for each metric, it becomes possible to calculate the precision and recall for the correct annotations. Following the definitions provided in the TypeWriter paper [31], precision is calculated as $prec = \frac{n_{corr}}{n_{all}}$, where n_{corr} is the number of correct predictions and n_{all} represents the total count of non-empty type slots. Similarly, recall is determined as $rec = \frac{n_{corr}}{|D|}$, where $|D|$ represents the total number of type slots.

4.4.4. Performance Evaluation

To address Research Question 2, it is crucial to monitor performance characteristics like speed and memory consumption. In addition to the list of metrics outlined in Section 4.4.2, more metrics are gathered for each file of the provided project during the runtime of the PyHintSearch tool. These additional metrics are:

- Time taken by Pyright
- Time taken by PyHintSearch
- Average time per PyHintSearch evaluated type slot
- Total time taken

- Peak memory usage of Pyright
- Peak memory usage of PyHintSearch

4.4.5. Call Graph Evaluation

For the evaluation of Research Question 3, the impact of type annotations on downstream program analysis is assessed. In this evaluation, call graphs are utilised based on the hypothesis that type annotations in Python code provide additional information to the call graph generation tool. This additional information potentially enables more precise specification of the edges in a call graph, allowing for the removal of these edges with greater accuracy, thereby enhancing its precision. These enhancements would consequently improve static analysis given the relevance of call graphs in that context. The call graph generation tool used in this thesis is provided by a company called Endor Labs[7]. Their proprietary tool is said to be superior to alternatives like PyCG [38] and Jarvis [52], generating more precise call graphs.

To start the evaluation, a baseline call graph is generated from the original Python code of a user-provided project. This project may already contain some type annotations or none at all. The Python code is processed by the Endor Labs call graph generator, which produces a JSON file detailing the nodes and edges that make up the call graph. Notably, Endor Labs' tool leverages Pyright to infer type annotations in addition to collecting them from Python's typeshed repository [32] prior to generating the call graph. Thus, the baseline call graph is created from code already annotated with Pyright annotations.

Next, the PyHintSearch tool is applied to the user-provided project, aiming to add as many validated type annotations into the code as possible. This process is done for the top-1, top-3 and top-5 predictions. The resulting annotated source code is then given to Endor Labs' call graph generator, producing call graphs corresponding to the top-1, top-3 and top-5 predictions, respectively. These call graphs are then compared to the baseline to quantify the number of similar, added and removed edges, enabling the calculation of the precision and recall between the two graphs. These values contribute to answering the question of how type annotations impact downstream program analysis. Ideally, the number of removed edges is relatively high, indicating that the additional information provided by type annotations enhances the precision of the call graph.

4.4.6. Experimental Setup

The PyHintSearch tool was evaluated on three Google Cloud Compute Engine VM instances, each equipped with 2 vCPUs, 1 core and 8 GB of memory. While a single instance could have sufficed, using three instances enabled the parallel execution of PyHintSearch, thus speeding up the collection of results.

[7]`https://www.endorlabs.com`

5 Results

This chapter presents the results obtained from the PyHintSearch tool utilising both the Type4Py and TypeT5 models. Results related to the coverage and correctness of the added type annotations are shown, alongside performance metrics for the tool. Additionally, information related a call graph that is generated with and without added type annotations by Type4Py, is also shown.

5.1. Type4Py

Type4Py is the default machine learning model of the PyHintSearch tool for generating predictions which are searched. This search process is executed across 16 projects, which are showcased in Table 5.1 with their corresponding commit hash for potential reproduction.

Table 5.1: All projects used for the evaluation of PyHintSearch with the Type4Py model.

Project	Commit hash	Project	Commit hash
Black	632f44b	Html5lib	82c2599
Bleach	55d9d60	Matplotlib	26832df
Braintree	282cb0d	Pandas	94d575a
Colorama	1368087	Pillow	e478775
Dateparser	1d4b058	Redis	1a7d474
Django	0630ca5	Requests	96b22fa
Exifread	d60f18d	Seaborn	b95d6d1
Flask	94e80b3	Stripe	66c96bf

These projects are chosen as they represent a diverse selection of code bases, ranging from small to large, including both popular and lesser-known projects. They also have already-existing type stubs, which offer additional information to PyHintSearch and enable comparisons to be made against the added predicted types.

5.1.1. Coverage

To answer Research Question 1 concerning the overall improvement of additionally determined type annotations, several statistics from all projects are tracked. For example, the average number of type slots across all files is logged and, since many projects already contain some type annotations, it means that many of these slots have been filled in already. Therefore, knowing the average number of remaining fillable type slots gives an indication of how many type slots can be evaluated by both Pyright and PyHintSearch. After running Pyright, some slots get filled with a type annotation and similarly for PyHintSearch. Hence, to elaborate on the overall improvement, it is informative to observe the number of type slots left unfilled after running either tool. These results, depicted in Table 5.2, reveal that initially, there are 17.09 fillable type slots across all projects. Then after Pyright's type propagation, 13.40 slots remain unfilled. Finally, after running PyHintSearch, this number decreases considerably to 5.03 unfilled type slots, considering the top-1 predictions of the type inference machine learning model.

Table 5.2: Type slots information for all projects.

Slots		Mean type slots
Total type slots		41.10
Fillable type slots		17.09
Unfilled type slots after Pyright propagation		13.40
	Top-1	5.03
Unfilled type slots after PyHintSearch	Top-3	5.18
	Top-5	5.26

Looking at these results from another perspective, it's interesting to not only examine the information related to type slots, but also the actual number of annotations present on average in a project. Table 5.3 provides this perspective. Initially, the mean number of annotations in a project is 24.01. However, if Pyright is used to propagate additional type annotations, this value increases to 27.70. Hence, Pyright is capable of determining, on average, an additional 3.69 type annotations, which represents 21.61% of the fillable type slots. Following Pyright, PyHintSearch is ran, resulting in an additional 8.36 type annotations when considering the top-1 predictions, thus ending up at an average of 36.07 validated type annotations per project file. This translates to an additional 48.94% of the fillable type slots being filled by PyHintSearch. However, more importantly, the machine learning model and combinatorial search that make up PyHintSearch ultimately enhances the Pyright inferred annotations by an additional 62.45%. For the top-3 and top-5 scenarios, this increase is slightly lower at 61.32% and 60.75%, respectively.

Table 5.3: Number of annotations for all projects. All values are mean values.

Phase		Annotations	Added annotations	Added annotations (%)	Added annotations after Pyright (%)
Initial		24.01	-	-	-
After Pyright propagation		27.70	3.69	21.61	-
	Top-1	36.07	8.36	48.94	**62.45**
After PyHintSearch	Top-3	35.92	8.21	48.06	**61.32**
	Top-5	35.84	8.14	47.62	**60.75**

To visually demonstrate the improved coverage by PyHintSearch, take a look at Figure 5.1. In this histogram, each file across all projects is categorised by the number of unfilled type slots, illustrating the initial distribution. Then, after running PyHintSearch, more annotations get added, resulting in fewer unfilled type slots, thus showing a shift in the distribution.

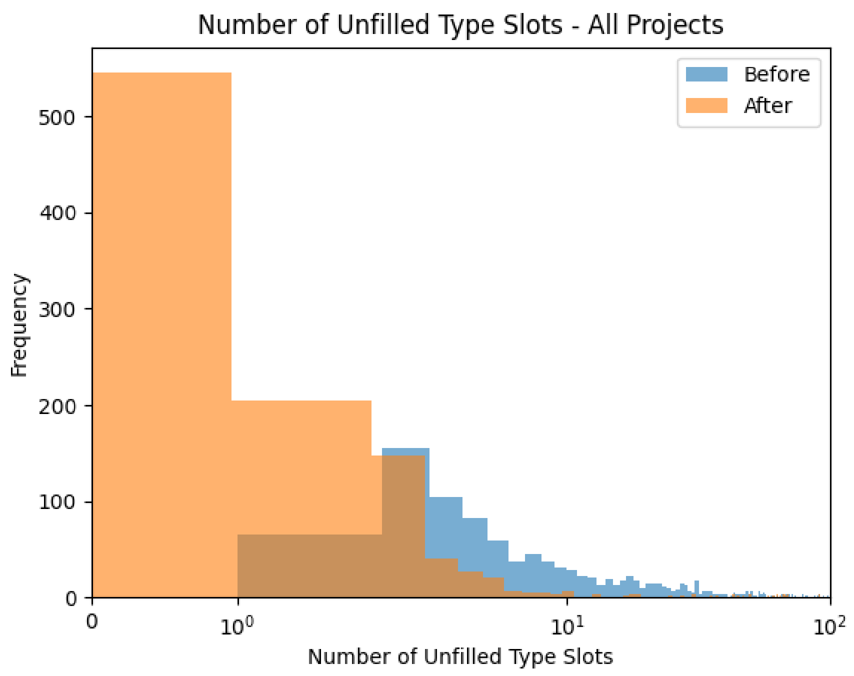


Figure 5.1: Distribution of unfilled type slots before and after PyHintSearch. The distribution is based on the number of unfilled type slots for each file in all the evaluated projects. The x-axis uses the symlog scale.

Of all the type annotations in the projects, it is also insightful to understand their categorisation into ubiquitous, common and rare types. Moreover, this categorisation can be specified even further since PyHintSearch only works for function arguments and return types. Table 5.4 offers an absolute overview of this categorisation, while Table 5.5 provides relative percentages. Interesting to note is that for the initial types, the number of rare annotations is larger than the number of common annotations, as developers often add project-specific type annotations which are not in the top-100 most frequent types. Conversely, Type4Py's determined type annotations do follow the expected categorisation of ubiquitous, common and rare. Both tables also point out Pyright's inability to infer type annotations for function arguments, highlighting the value of using a machine learning model to address this gap.

Table 5.4: Absolute categorisation of ubiquitous, common and rare type annotations for all projects.

Types		Ubiquitous		Common		Rare	
		Args	Returns	Args	Returns	Args	Returns
Initial types		8456	4815	1959	883	4732	3762
Pyright propagated		0	2132	0	512	0	656
	Top-1	5920	1746	975	122	414	28
PyHintSearch added	Top-3	5767	1702	942	140	441	49
	Top-5	5700	1676	935	157	438	51

Table 5.5: Relative categorisation of ubiquitous, common and rare type annotations for all projects.

Top-n	Types	% Ubiquitous		% Common		% Rare	
		Args	Returns	Args	Returns	Args	Returns
Top-1	Initial types	22.8	13.0	5.3	2.4	12.8	10.0
	Pyright propagated	0.0	5.7	0.0	1.4	0.0	1.8
	PyHintSearch added	16.0	4.7	2.6	0.3	1.1	0.1
Top-3	Initial types	22.9	13.0	5.3	2.4	12.8	10.2
	Pyright propagated	0.0	5.8	0.0	1.4	0.0	1.8
	PyHintSearch added	15.6	4.6	2.5	0.4	1.2	0.1
Top-5	Initial types	22.9	13.1	5.3	2.4	12.8	10.2
	Pyright propagated	0.0	5.8	0.0	1.4	0.0	1.8
	PyHintSearch added	15.5	4.5	2.5	0.4	1.2	0.1

5.1.2. Correctness

While the correctness of the type annotations added by PyHintSearch relies entirely on the precision of the underlying machine learning model, it remains valuable to assess how many of these annotations match with the original types. To achieve this, all project files get stripped from their type annotations, followed by the execution of both Pyright and PyHintSearch on the stripped files. Now, the newly added type annotations from either method can be compared to the original annotations present in the files. As described in Section 4.4.3, there are two accuracy metrics used to perform this comparison, namely the adjusted accuracy and base accuracy. Across all projects, the mean number of original type annotations is 17.02, serving as the baseline for the comparison.

Both Pyright and the PyHintSearch tool are capable of adding new annotations to the stripped projects. However, PyHintSearch uses Pyright alongside a validated combinatorial search, thus enabling it to annotate more files than Pyright could on its own. In mathematical terms, the output of the PyHintSearch tool is the superset of Pyright's output. Because of this, in order to make fair correctness comparisons between the two tools, only the files which Pyright is able to annotate are considered for the results in Table 5.6.

Table 5.6: Correctness of the added type annotations for all projects, excluding Python dunder methods. All values are mean values.

Accuracy	Phase	Original	Added	Correct	Incorrect	Precision	Recall
Adjusted	After Pyright	17.02	3.74	1.43	15.59	0.38	0.08
	After Top-1	17.02	12.40	4.46	12.56	0.36	0.26
	After Top-3	17.02	12.14	4.32	12.70	0.36	0.25
	After Top-5	17.02	12.11	4.32	12.70	0.36	0.25
Base	After Pyright	17.02	3.74	2.18	14.84	0.58	0.13
	After Top-1	17.02	12.40	5.31	11.71	0.43	0.31
	After Top-3	17.02	12.14	5.19	11.83	0.43	0.31
	After Top-5	17.02	12.11	5.19	11.83	0.43	0.30

In terms of adjusted accuracy, Pyright typically adds an average of 3.74 type annotations, with only 1.43 of them being correct. Therefore, 15.59 annotations are incorrect, either failing to match the original types or being absent altogether. On the other hand, the PyHintSearch tool, using top-1 predictions, demonstrates better performance by adding 12.40 extra types, of which 4.46 are correct. While this suggests a greater number of type annotations integrated into the code, a significant portion still does not match the original annotations.

Regarding base accuracy, the number of Pyright added type annotations is the same, yet 2.18 more annotations are deemed correct due to the relaxed constraint of only matching the base type. Similarly, across all top-*n* predictions, the number of correct type annotations increases. Specifically, top-1 predictions achieve correctness for 5.31 types, but still leave 11.71 annotations unmatched or missing.

5.1.3. Performance

Time Performance metrics provide insights into the speed, memory usage and scalability of the developed tool. There are two distinct steps that take place when adding annotations, namely the Pyright step and the PyHintSearch step. The combined time that the Pyright step takes is 4,346.51 seconds or approximately 1.25 hours across all files from all the projects. Meaning that for each file, this step takes on average 3.93 seconds.

Regarding the PyHintSearch step, the time taken is primarily dependent on the top-*n* predictions that are explored during the validated combinatorial search. If the majority of machine learning-predicted annotations are valid according to the feedback from Pyright, then the type slots can be filled in linear time. However, if the need for backtracking arises, the number of top-*n* annotations significantly affects the time required. Continuously trying only a single type annotation before filling in a blank or continuing to the next slot goes much faster than performing the same process for five annotations. The only reason that the times in Table 5.7 do not exhibit exponential growth for larger top-*n* values is due to the tool's 5-minute timeout per file. This limitation restricts the combinatorial search from taking hours before finding a valid combination of type annotations, but also results in no combination being found when the timeout is reached. However, that is the trade-off for having a practically applicable tool instead of waiting hours, if not days, searching the exponential space of combinations.

Nevertheless, automating annotations for 16 projects during the PyHintSearch step consumes 44,003.46 seconds (~12.25 hours) for top-1 predictions, increasing to 52,093.32 seconds (~14.5 hours) for top-5 predictions. Despite the time investment, this automated approach is still much faster than manual annotation by a developer, particularly for large projects like Django and Pandas.

Table 5.7: The execution time of each step in the tool.

Step	Top-*n*	Mean time per file (s)	Total time (s)	Total time (h)
Pyright		3.93	4,346.51	~1.25
	Top-1	39.75	44,003.46	~12.25
PyHintSearch	Top-3	45.74	50,634.29	~14
	Top-5	47.06	52,093.32	~14.5
	Top-1	44.38	49,125.39	~13.75
Total*	Top-3	50.23	55,603.95	~15.5
	Top-5	51.56	57,079.13	~16

* Total time of the tool, thus including other minor computations in addition to the Pyright and PyHintSearch steps.

Memory As for memory usage, the maximum values that have been observed during the execution of the tool are 23.67 megabytes for the Pyright step and 119.99 megabytes for the PyHintSearch step. These values are relatively small and most computers should be able to handle this amount of memory usage. It should be noted, however, that PyHintSearch relies on a machine learning model accessible via a Docker container for its predictions. This container takes up several gigabytes of memory, thus it's advisable to allocate a minimum of 8 gigabytes or more of memory to ensure smooth operation.

Scalability The scalability of the PyHintSearch tool is constrained by several factors to ensure practical feasibility. Firstly, there is a hard limit of 100 fillable type slots after the Pyright step, that the validated combinatorial search tries to find a valid combination of type annotations for. This limitation is set because larger files often have more interdependent functions, which increases the likelihood of requiring backtracking. Such backtracking results in an exponential increase in the number of combinations to be explored, often leading to the tool reaching its timeout threshold of five minutes. This is the second limitation as the search is stopped and no new annotations are added to the file. Therefore, while removable, without this timeout limitation, the time for finding a valid combination would scale exponentially, rendering the tool impractical for usage.

5.1.4. Call Graphs

Following the execution of PyHintSearch, all 16 projects are enriched with type annotations from the top-1, top-3 and top-5 predictions. These annotated project files are then processed by Endor Labs' call graph generation tool to generate corresponding call graphs. These call graphs are compared to the baseline call graph, generated from the original user-provided project, to examine the differences in edges between the two graphs. Table 5.8 provides an overview of each project's call graph, detailing the number of similar, added and removed edges.

Table 5.8: Overview of the number of edges present in a call graph.

Project	Existing CG	New CG	Same	Added	Removed	Precision	Recall
Black							
Top-1	2370	2368	2368	0	2	1	1
Top-3	2370	2368	2368	0	2	1	1
Top-5	2370	2368	2368	0	2	1	1
Bleach							
Top-1	816	669	617	52	199	0.92	0.76
Top-3	816	669	617	52	199	0.92	0.76
Top-5	816	669	617	52	199	0.92	0.76
Braintree							
Top-1	6027	5719	5603	116	424	0.98	0.93
Top-3	6027	5708	5551	157	476	0.97	0.92
Top-5	6027	5708	5551	157	476	0.97	0.92
Colorama							
Top-1	181	181	181	0	0	1	1
Top-3	181	181	181	0	0	1	1
Top-5	181	181	181	0	0	1	1
Dateparser							
Top-1	2925	2975	2722	253	203	0.91	0.93
Top-3	2925	3026	2789	237	136	0.92	0.95
Top-5	2925	3026	2789	237	136	0.92	0.95
Django							
Top-1	259519	258950	255549	3401	3970	0.99	0.98
Top-3	259519	263875	255587	8288	3932	0.97	0.98
Top-5	259519	264252	255581	8671	3938	0.97	0.98
Exifread							
Top-1	414	416	413	3	1	0.99	1
Top-3	414	416	413	3	1	0.99	1
Top-5	414	416	413	3	1	0.99	1

Continued on next page

Table 5.8 – continued from previous page

Project	Existing CG	New CG	Same	Added	Removed	Precision	Recall
Flask							
Top-1	1587	1631	1583	48	4	0.97	1
Top-3	1587	1631	1583	48	4	0.97	1
Top-5	1587	1631	1583	48	4	0.97	1
Html5lib							
Top-1	5750	5391	5331	60	419	0.99	0.93
Top-3	5750	5351	5292	59	458	0.99	0.92
Top-5	5750	5351	5292	59	458	0.99	0.92
Matplotlib							
Top-1	61701	62957	59974	2983	1727	0.95	0.97
Top-3	61701	63039	60048	2991	1653	0.95	0.97
Top-5	61701	63045	60049	2996	1652	0.95	0.97
Pandas							
Top-1	87585	87585	87585	0	0	1	1
Top-3	87585	87585	87585	0	0	1	1
Top-5	87585	87585	87585	0	0	1	1
Pillow							
Top-1	12877	12705	12341	364	536	0.97	0.96
Top-3	12877	12724	12341	383	536	0.97	0.96
Top-5	12877	12724	12341	383	536	0.97	0.96
Redis							
Top-1	14370	14140	14008	132	362	0.99	0.97
Top-3	14370	14140	14008	132	362	0.99	0.97
Top-5	14370	14089	14038	51	332	1	0.98
Requests							
Top-1	2306	2166	1956	210	350	0.9	0.85
Top-3	2306	2228	2062	166	244	0.93	0.89
Top-5	2306	2228	2062	166	244	0.93	0.89
Seaborn							
Top-1	24970	24529	24281	248	689	0.99	0.97
Top-3	24970	24664	24429	235	541	0.99	0.98
Top-5	24970	24875	24655	220	315	0.99	0.99
Stripe							
Top-1	55027	55027	55027	0	0	1	1
Top-3	55027	55027	55027	0	0	1	1
Top-5	55027	55027	55027	0	0	1	1

A noteworthy observation is related to the projects Black, Colorama, Exifread, Pandas and Stripe. These projects stand out because of their minimal changes in added or removed edges, with many edges matching exactly with the baseline call graph. This can be attributed to the extensive integration of type annotations from the corresponding type stubs in the code, leading to a limited number of available type slots for PyHintSearch to fill. Therefore, running PyHintSearch over these projects has minimal impact on the number of added annotations, resulting in nearly identical call graphs.

Moreover, several interesting but also unexpected results can be seen for the Dateparser, Django, Flask and Matplotlib projects. For these projects, the number of edges added to the call graph exceeds the number of removed edges. Conversely, in the remaining projects, the number of removed edges is larger than the number of added edges. This supports the hypothesis that additional type information contributes to a more precise call graph. However, to gain a more thorough understanding of why call graphs have added and removed edges, several edges from different projects were examined. This examination includes the project name to which the edge belongs, the name of the examined target edge, a classification of whether source edges were added or removed for the target edge, and an explanation of my reasoning behind the addition or removal of the source edges. The results of this examination are shown in Table 5.9, which can be found on the next page.

Table 5.9: Examination of the call graph edges and their corresponding source edges.

Project	Edge	Change	Explanation
Bleach	`sanitizer.BleachSanitizerFilter.sanitize_stream`	Removed and added sources	The `token_iterator` parameter of the `sanitize_stream` function was annotated with `List[object]` and the return type as `Generator`, removing 39 generic source edges related to `__next__` and adding two specific source edges related to `builtins.list`.
Bleach	`html5lib_shim.match_entity`	Added sources	The `stream` parameter of the `match_entity` function was annotated incorrectly with `TextIO`. In the function body, this parameter is cast to a `list`, but because of the `TextIO` type annotation, 10 generic `pop` source edges are added, since it is overridden and not regarded as a list anymore.
Dateparser	`utils.localize_timezone`	Removed sources	The parameters of the `localize_timezone` function were annotated with `str`, removing 12 source edges related to generic `replace` methods.
Dateparser	`languages.dictionary.Dictionary.are_tokens_valid`	Removed and added sources	The `tokens` parameter of the `are_tokens_valid` method was annotated with `str`, removing 41 generic source edges related to `__next__` and `isdigit` and adding two specific source edges related to `builtins.str`.
Html5lib	`_trie.py.Trie.__init__`	Removed sources	The `data` parameter of the `__init__` method was annotated with `dict`, removing 13 source edges related to generic `keys` functions.
Html5lib	`_tokenizer.HTMLTokenizer.__iter__`	Added sources	The `__iter__` method references `self.stream` which is a `HTMLInputStream` object. The `__init__` method of this class is annotated with `Union[HTMLUnicodeInputStream, HTMLBinaryInputStream]`, but those classes are defined later. Therefore, its `error` property, which was seen as a list, is now regarded as `Any` and 13 generic `pop` source edges are added.
Pillow	`ImageFile._save`	Removed sources	The `im` parameter of the `_save` function was annotated with `Image`, removing 22 source edges related to generic `load` methods.
Matplotlib	`tri._tripcolor.tripcolor`	Added sources	The `ax` parameter of the `tripcolor` function was annotated incorrectly with `bool`, adding 1 source edge named `pip._vendor.rich.table.Table.grid` which is also incorrect as it should have referred to the local `axis.grid` function.

5.2. TypeT5

TypeT5 is an alternative machine learning model that can be used by the PyHintSearch tool for predicting type annotations. While PyHintSearch can theoretically use any model that predicts multiple type annotations, TypeT5's implementation yields only a single prediction. Therefore, PyHintSearch resorts to its top-1 prediction during the combinatorial search process. Furthermore, TypeT5 uses all project files at once to improve the precision of its type annotation predictions. However, its biggest drawback is that it is very slow in predicting types for all these files when not utilising a GPU. This long prediction time lead to a reduced number of projects for which results could be obtained. Instead of gathering results for the original 16 projects, the results are limited to just 4, namely, Bleach, Colorama, Requests and Seaborn. These are presented in Table 5.10.

Table 5.10: All projects used for the evaluation of PyHintSearch with the TypeT5 model.

Project	Commit hash
Bleach	55d9d60
Colorama	1368087
Requests	96b22fa
Seaborn	b95d6d1

Similarly to the Type4Py section above, the coverage, correctness and performance results are gathered for the PyHintSearch tool that utilises the TypeT5 model. Given that TypeT5 generates only one prediction, it is compared to the top-1 predictions of the Type4Py model to make a fair comparison and to assess the impact of the underlying machine learning model on the effectiveness of the combinatorial search.

5.2.1. Coverage

To compare TypeT5 against Type4Py on type annotation coverage of the four projects, a general overview of information about the type slots is required. This overview is displayed in Table 5.11, showing that the mean number of total type slots is 43.02, with 20.09 slots yet to be filled. Upon running Pyright across the four TypeT5 projects, this value decreases to 16.00 unfilled type slots. A further improvement to this value is achieved by running PyHintSearch. Utilising the Type4Py model decreases the number of unfilled type slots to 5.58, whereas with the TypeT5 model, this number drops even further to 3.21 unfilled type slots.

Table 5.11: Type slots information for all projects processed by PyHintSearch with TypeT5.

Slots	Model	Mean type slots
Total type slots		43.02
Fillable type slots		20.09
Unfilled type slots after Pyright propagation		16.00
Unfilled type slots after PyHintSearch	Type4Py	5.58
	TypeT5	3.21

Instead of looking only at type slot information, the number of type annotations in the code can also be examined together with their increase after inferring types annotations. Table 5.12 illustrates that initially, the mean number of annotations is 22.93, which rises to 27.02 after Pyright's propagation. Pyright, therefore, adds an average of 4.09 type annotations, which is a 20.35% increase based on the number of fillable slots. PyHintSearch improves upon this value with Type4Py and TypeT5, adding 10.42 and 12.79 annotations respectively. Therefore, Type4Py can fill an additional 51.88% of the fillable type slots, while TypeT5 fills 63.67%. More importantly, Type4Py improves upon Pyright by 65.35%, but this remarkable achievement is surpassed by the utilisation of the TypeT5 model, realising an improved score of 79.93%. This clearly indicates that TypeT5 predicts more accurate, or at least more valid type annotations according to the Pyright type checker, which are accepted in the validated combinatorial search compared to Type4Py.

Table 5.12: Number of annotations for all projects processed by PyHintSearch with TypeT5. All values are mean values.

Phase	Model	Annotations	Added annotations	Added annotations (%)	Added annotations after Pyright (%)
Initial		22.93	-	-	-
After Pyright propagation		27.02	4.09	20.35	-
After PyHintSearch	Type4Py	37.44	10.42	51.88	**65.35**
	TypeT5	39.81	12.79	63.67	**79.93**

The initial and added type annotations can all be categorised into ubiquitous, common and rare annotations. Table 5.13 shows the absolute values for this categorisation, while Table 5.14 presents the relative values. It's worth noting that PyHintSearch with the TypeT5 model is able to add more rare annotations compared to the Type4Py model. Specifically, while only 1.2% of the added annotations with Type4Py are rare, this value increases to 7.6% when using TypeT5. Furthermore, the percentage of common types is also higher for TypeT5, showcasing its capacity to predict more infrequent type annotations compared to Type4Py.

Table 5.13: Absolute categorisation of ubiquitous, common and rare type annotations for all projects processed by PyHintSearch with TypeT5.

Types	Model	Ubiquitous		Common		Rare	
		Args	Returns	Args	Returns	Args	Returns
Initial types		362	215	142	50	283	174
Pyright propagated		0	141	0	28	0	14
PyHintSearch added	Type4Py	327	93	134	12	21	2
	TypeT5	283	96	150	30	122	41

Table 5.14: Relative categorisation of ubiquitous, common and rare type annotations for all projects processed by PyHintSearch with TypeT5.

Model	Types	% Ubiquitous		% Common		% Rare	
		Args	Returns	Args	Returns	Args	Returns
Type4Py	Initial types	18.1	10.8	7.1	2.5	14.2	8.7
	Pyright propagated	0.0	7.1	0.0	1.4	0.0	0.7
	PyHintSearch added	16.4	4.7	6.7	0.6	1.1	0.1
TypeT5	Initial types	17.0	10.1	6.7	2.3	13.3	8.2
	Pyright propagated	0.0	6.6	0.0	1.3	0.0	0.7
	PyHintSearch added	13.3	4.5	7.0	1.4	5.7	1.9

5.2.2. Correctness

To demonstrate the superior accuracy of the predicted type annotations by TypeT5 compared to those of Type4Py, the correctness of the added annotations is examined in relation to the original ones. Table 5.15 reveals that across the four projects analysed by TypeT5, an average of 26.20 annotations were initially present in the codebase. After stripping these out and running Pyright, an average of 5.85 annotations were added. According to the adjusted accuracy metric, 2.40 of these annotations were deemed correct, increasing to 3.50 for the base accuracy.

Table 5.15: Correctness of the added type annotations for all projects processed by PyHintSearch with TypeT5, excluding Python's dunder methods. All values are mean values.

Accuracy	Phase	Original	Added	Correct	Incorrect	Precision	Recall
Adjusted	After Pyright	26.20	5.85	2.40	23.80	0.38	0.07
	After Type4Py	26.20	16.48	5.45	20.75	0.33	0.21
	After TypeT5	26.20	17.70	9.10	17.10	0.51	0.35
Base	After Pyright	26.20	5.85	3.50	22.70	0.55	0.11
	After Type4Py	26.20	16.48	6.68	19.53	0.41	0.25
	After TypeT5	26.20	17.70	10.63	15.58	0.60	0.41

After running PyHintSearch with Type4Py, 16.48 annotations were added. Of these annotations, 5.45 were deemed correct based on the adjusted accuracy and 6.68 based on the base accuracy. Although there was a decrease in precision compared to Pyright, there was an increase in recall. Conversely, the more accurate type predictions by TypeT5 led to the addition of 17.70 types, with 9.10 deemed correct based on the adjusted accuracy and 10.63 based on the base accuracy. Looking at these results, it can be seen that TypeT5 has increased precision and recall compared to both Pyright and Type4Py. This shows the superiority of a model that offers more accurate type annotations to be used in the combinatorial search.

5.2.3. Performance

Time Although TypeT5's type predictions are more accurate than those of Type4Py, it also takes significantly longer to compute them. The time taken for each project can be found in Table 5.16 and shows that small projects such as Bleach and Colorama require approximately 5 minutes for the computation of the predictions, whereas for a medium-sized project like Seaborn, the time increased drastically to almost 3 hours. Due to this considerable increase in time, large projects like Django, Matplotlib, Pillow or Pandas were not attempted, as their predictions would likely require numerous hours to complete before the combinatorial search could even start.

Table 5.16: Time taken by TypeT5 to predict type annotations for each project which are used in the validated combinatorial search.

Project	Total time (s)	Total time (min)
Bleach	288.34	~5
Colorama	279.65	~4.5
Requests	1525.50	~25.5
Seaborn	10517.79	~175.5
Total	12611.28	~210.5

Nevertheless, the four projects that TypeT5 was run over, can be used to compare the speed of TypeT5 to both Pyright and Type4Py. The results of execution time are shown in Table 5.17 which displays that the Pyright step took 155.44 seconds for all files. Comparatively, PyHintSearch utilising the Type4Py model took 2,697.86 seconds for all files, whereas TypeT5 took much longer at 14,634.88 seconds.

Table 5.17: The execution time of each step in the tool.

Step	Model	Mean time per file (s)	Total time (s)	Total time (min)
Pyright		2.73	155.44	~2.5
PyHintSearch	Type4Py	47.33	2,697.86	~45
	TypeT5	256.75	14,634.88	~244
Total[*]	Type4Py	51.05	2,910.02	~48.5
	TypeT5	260.16	14,829.16	~247

[*] Total time of the tool, thus including other minor computations in addition to the Pyright and PyHintSearch steps.

Memory On the smaller number of projects that were processed by TypeT5, Pyright's maximum memory usage was only 12.92 megabytes. However, when running the combinatorial search with predictions from Type4Py, the maximum memory usage rose to 107.65 megabytes and increased further to 120.42 megabytes when utilising the TypeT5 model.

Scalability The scalability of PyHintSearch using the TypeT5 model is similar to the results discussed in Section 5.1.3 regarding PyHintSearch with the Type4Py model. However, the main difference lies in TypeT5's ability to predict only a single type annotation, thus limiting the exponential growth during backtracking.

6 Discussion

In this chapter, the importance and consequences of the presented results are summarised. First, the main findings are discussed, followed by the limitations of the research and the PyHintSearch tool, and finishing with a threat to the validity of this thesis.

6.1. Main Findings

Coverage and Correctness This thesis demonstrates that PyHintSearch significantly improves the type annotation coverage of Python code by using both Pyright's static type inference and a validated combinatorial search strategy for machine learning-based predictions. PyHintSearch complements Pyright's static type inference effectively by adding 62.45% more annotations when using the Type4Py model and 79.93% for the TypeT5 model. This is a notable increase in the number of type annotations compared to using only Pyright. However, mismatches between the original annotations and those added by PyHintSearch highlight the challenge of achieving perfect correctness. When utilising the Type4Py model, approximately one-third of the type annotations match according to the adjusted accuracy metric, whereas for the TypeT5 model, this value increases to approximately half of the type annotations matching. Fortunately, any type inference machine learning model can by used by PyHintSearch as long as it can provide its predictions in a predefined JSON format. Therefore, further improvements in the accuracy of machine learning models can minimise the discrepancies between the original and PyHintSearch-added annotations leading to more reliable and correct annotations.

Most Effective Top-*n* Using the top-1 predictions of the predicted type annotations in the combinatorial search outperforms both the top-3 and top-5. The search with top-1 predictions fills more type slots, has slightly more correct type annotations, and is overall 1.75 hours faster compared to the top-3 and 2.25 hours faster compared to the top-5. The only slight downside of using top-1 is having a couple fewer rare type annotations added to the codebase, but that is negligible compared to the larger number of added ubiquitous annotations. The most likely reason for the top-1 performing better than the top-3 and top-5, which in essence both include the top-1 predictions, is the fact that if a file needs to be backtracked to find a valid combination of type annotations, the top-3 and top-5 can backtrack fewer levels up the search tree. This is because more annotations need to be tried per slot, leading to hitting the 5-minute timeout per file more frequently. Furthermore, if the 5-minute timeout is hit, no new annotations get added to the project code. The top-1 is thus able to backtrack more quickly and find a branch in the search tree that can progress the combinatorial search before hitting the 5-minute timeout, which leads to an increased number of type annotations.

If no timeout was present and all the time in the world was available, then the top-3 and top-5 would reach similar or better results. However, that is not the case as PyHintSearch needs to be feasible for real-world usage.

Performance The performance metrics gathered over sixteen projects indicate that PyHintSearch can, on average, annotate a project file in 39.75 seconds after the Pyright-inferred annotations were added. The total time for the PyHintSearch tool using the top-1 predictions was 49,125.39 seconds or approximately 13.75 hours to annotate all project files. This is much faster than a human could annotate these files by hand. Furthermore, the maximum memory usage that was measured for PyHintSearch was 119.99 megabytes, although this does exclude the Docker container for the machine learning model that needs to be run at the same time. The Docker container takes up multiple gigabytes of memory, thus a minimum of 8 gigabytes of memory is recommended.

Impact and Frequency of Backtracking It turns out that when backtracking is required for a file, the combinatorial search usually fails because it hits the timeout limit of 5 minutes. This is because the backtracking needs to go up multiple levels of the search tree to correct the annotation that is being a bottleneck lower down the tree. When backtracking, each branch in the search tree is tried, meaning an exponential number of combinations are searched before finding a combination of type annotations that can overcome the bottleneck. Fortunately, many files can be filled in linearly because either the type annotations predicted by the machine learning model are valid or the functions defined in the file are independent of one another thus avoiding the need of backtracking. Therefore, only a few files actually need to be backtracked and the search takes place in linear time. An example of successful backtracking to find a valid combination of type annotations can be found in Appendix A.

Call Graphs The results concerning call graphs confirm that incorporating type annotations into Python code enhances the accuracy of the generated call graph by providing additional information to the call graph generation tool. These annotations enable the removal of edges, resulting in a more precise call graph that is useful for static analysis purposes. For example, annotating a function parameter with the `dict` annotation resulted in the removal of 13 generic `keys` functions like, `typing.Mapping.keys`, `collections.OrderedDict.keys`, `xml.etree.ElementTree.Element.keys`, etc. Contrastingly, edges were also added to the call graphs of several projects. This typically occurred when an incorrect type annotation was found by PyHintSearch, leading to function parameters and variables being less specific in their types. For example, a variable could originally have been inferred as a `str` type, but is now inferred as an `Any` type due to the cascading effect of an incorrect parameter annotation. Interestingly, in some cases, the addition of edges was beneficial as they refined the call graph even further. For instance, 39 edges related to a generic `__next__` function were removed, while two specific edges related to `builtins.list` were added. These removed and added edges were the result of annotating a function parameter as `List`. Therefore, it can be concluded that type annotations do indeed impact downstream program analysis as they influence the generation of call graphs.

6.2. Limitations

The major limitation of the PyHintSearch tool arises when two functions are interdependent. If the first function is annotated incorrectly but still passes Pyright's validity check, it becomes a bottleneck for the second function. Resolving this issue requires backtracking and trying different annotations until the correct one is found. When these functions are closely located to one another in the codebase, backtracking does not need to go up many levels of the search tree to fix the incorrect type annotation. However, if they are distant, then there is a large number of levels in the tree which are searched exponentially. This usually leads to the 5-minute timeout being hit and no annotations being added.

Another limitation lies in the prediction time of the TypeT5 model. Despite its superior accuracy over Type4Py, TypeT5 is considerably slower. While Type4Py can predict type annotations for individual files, TypeT5 requires all project files as input for its predictions. This means that Type4Py can generate predictions for a file within seconds, whereas that is not the case for TypeT5. Even for a small project comprising around 10 Python files, TypeT5 takes approximately 5 minutes for its predictions. Scaling up to a medium-sized project of around 50 files extends the prediction time to nearly 3 hours, excluding the combinatorial search. Due to these extensive prediction times, the default model used for PyHintSearch during the thesis was Type4Py. Although less accurate, it took much less time to generate predictions, which allowed for more rapid prototyping of the search algorithm. Furthermore, if during development an exception was raised causing the combinatorial search to crash, the predictions needed to be recomputed again, which would have been impractical given TypeT5's lengthy prediction times.

6.3. Threat to Validity

A potential threat to the validity of this thesis is primarily related to the results obtained from PyHintSearch utilising the TypeT5 model. Given the considerable amount of time required for TypeT5 predictions, only four projects were evaluated, in contrast to the 16 projects assessed with the Type4Py model. This discrepancy in the number of projects may lead to overly optimistic results regarding the effectiveness of PyHintSearch with TypeT5. Although TypeT5's performance is compared against Type4Py on the same files, a broader evaluation across more projects would have provided a more representative assessment, similar to the Type4Py evaluation detailed in Section 5.1.

7 Conclusion and Future Work

This thesis discussed the implementation and usage of a validated combinatorial search for probabilistic type inference models. This search contributes to enhanced type annotation coverage of Python code which improves static analysis. This enables more accurate error detection during the software development process, leading to increased code quality and reliability.

This chapter concludes this thesis by revisiting the research questions introduced in Chapter 1 and discussing promising directions for future work.

7.1. Conclusion

The PyHintSearch tool was developed to search for a valid combination of predicted type annotations from a machine learning type inference model. Type4Py and TypeT5 are two of these models used in this thesis. Especially the effectiveness and efficiency of PyHintSearch are evaluated in terms of coverage, performance and practical usage. Through the analysis of the collected results obtained from running PyHintSearch across several projects, the following three research questions have been answered:

RQ1 (Coverage): What is the overall increase in the number of type annotations when combining static type inference with a machine learning model?
From an assessment of 16 projects, it was observed that the Pyright static type checker was able to add 21.61% more type annotations to the initially provided code. Running PyHintSearch with the Type4Py model thereafter, yielded an additional 48.94% of validated type annotations when focusing on the top-1 predictions. These top-1 predictions were preferred since they added more type annotations to the code compared to the top-3 and top-5 predictions. Ultimately, the machine learning model complemented static type inference by incorporating 62.45% more annotations than Pyright could have on its own. However, the precision of the added type annotations is only 0.36, indicating that a significant portion of validated type annotations were incorrect. Fortunately, this is a limitation that can be addressed with a more accurate model.

TypeT5 represents such a more accurate model, therby motivating the evaluation of PyHintSearch with it. Despite being tested on fewer projects than the Type4Py model, the results are promising. Pyright inferred 20.35% more type annotations for the initial project, while PyHintSearch using TypeT5 further improved upon this by filling an additional 63.67% of fillable type slots. Consequently, PyHintSearch complemented Pyright with 79.93% more validated type annotations. Moreover, the precision increased to 0.49, showing that PyHintSearch becomes better at annotating Python code by using a more accurate model.

RQ2 (Performance): What are the performance characteristics, in terms of speed, memory usage and scalability, of the PyHintSearch tool?
To annotate all projects with validated type annotations, the Pyright step requires approximately 1.25 hours to complete, while the PyHintSearch step takes about 12.25 hours. Considering that this process annotates over 1000 files, it significantly outpaces the manual efforts of a developer.

Memory usage remains fairly limited, with Pyright using a maximum of 23.67 megabytes and PyHintSearch requiring a maximum of 119.99 megabytes for its top-1, top-3 and top-5 predictions.

The scalability of the PyHintSearch tool is dependent on the accuracy of the type predictions generated by the used machine learning type inference model. If the predicted type annotations are reasonably accurate, the type slots can be filled linearly with a maximum of n attempts per slot based on the top-n predictions. However, if the combinatorial search encounters an incorrectly defined function definition and needs to backtrack, it explores all branches of the search tree, resulting in an exponential execution time. To mitigate this, a built-in 5-minute timeout per file ensures that the total execution time remains manageable, regardless of the number of levels that require backtracking in the search tree. Furthermore, files with more than 100 fillable type slots are excluded from the combinatorial search due to the increased risk of getting stuck in backtracking caused by interdependent functions.

RQ3 (Use case): How does the presence of type annotations impact downstream program analysis?
Type annotations in Python code provide additional information to Endor Labs' call graph generation tool. By specifying the type annotations for function parameters and return types, the tool can remove edges from the call graph related to generic functions. Additionally, this information can be used to add more precise edges related to specific function calls. The larger the number of correct type annotations that are added into the source code, the more precise the call graph becomes, as they enable the tool to identify and eliminate irrelevant calls. For several projects, the addition of type annotations resulted in the call graph being reduced by hundreds of edges. Conversely, incorrect type annotations led to the addition of edges in the call graph. This occurred because the inferred types for variables within a function were overridden by the cascading effect of an incorrectly annotated function parameter. Consequently, this led to less precise variable types, which then resulted in the addition of more generic edges in the call graph. Overall, it can be concluded that the presence of type annotations influences the generation of call graphs by guiding the tool in making decisions about which edges to include or exclude. Ultimately, this contributes to a more precise call graph when type annotations are correct, thus impacting downstream program analysis.

7.2. Future Work

Based on insights gained from implementing PyHintSearch and the results from the evaluation, several directions are suggested for future work. These directions are aimed at another approach for enhancing type annotation coverage and further refinement of the PyHintSearch tool.

Genetic algorithm The core concept behind PyHintSearch involves constructing a search tree and applying a depth-first search method to identify a valid combination of type annotations, guided by feedback from the Pyright type checker. However, there exist alternative methods designed to navigate through the exponential space of possibilities in search of a valid combination. One interesting approach involves the usage of a genetic algorithm to create a random population of type predictions from the machine learning model. This population undergoes modifications through processes such as crossover or mutation and is evaluated against a fitness function. The most promising individuals (i.e., combinations of type annotations) are kept and new offspring are created. This iterative process continues until the algorithm converges, ultimately returning the highest-rated combination of type annotations.

Partial file changes To get Pyright's feedback on the validity of a newly added type annotation, a message is sent via the Language Server Protocol to Pyright's language server. This message denotes that the entire content of a file has been changed, rather than only the specific line where the type annotation is inserted. While not certain, it is expected that requesting feedback solely on such a minor alteration could potentially speed up the retrieval of Pyright's feedback, compared to requesting feedback on the entire file content.

Decreased memory usage For each file, the validated combinatorial search uses an array that stores complete LibCST trees. The larger the file, the larger the size of this array, because of the increased size of the trees that it stores, resulting in greater memory consumption. The only time these trees are used is to revert to a previous state when backtracking. However, a more efficient approach involves storing only the modifications required to change the original tree. This method reduces memory usage, as the original LibCST tree can be directly modified based on the annotations needed at a specific phase in the combinatorial search.

Minimum probability threshold The machine learning model used by PyHintSearch is able to return multiple type annotations, along with their associated probabilities. These annotations are then searched during the combinatorial search. However, this approach also allows for the inclusion of type annotations with very low probabilities of correctness. For instance, if a type annotation has a probability of only 10%, it is still considered in the combinatorial search when taking a large number of top-*n* predictions. This scenario could potentially lead to the insertion of type annotations into the source code that are considered valid by the static type checker but are not optimal in practice. In fact, it is better to refrain from adding an annotation than to include one that is incorrect in the source code. Therefore, future work could implement a minimum probability threshold that a type annotation must meet before being considered in the combinatorial search.

Bibliography

[1] Miltiadis Allamanis et al. “Typilus: neural type hints”. In: ACM, June 2020, pp. 91–105. ISBN: 9781450376136. DOI: 10.1145/3385412.3385997.

[2] Paul Anderson. “The Use and Limitations of Static-Analysis Tools to Improve Software Quality”. In: *CrossTalk: The Journal of Defense Software Engineering* 21.6 (2008), pp. 18–21.

[3] Nathaniel Ayewah et al. “Using Static Analysis to Find Bugs”. In: *IEEE Software* 25 (5 Sept. 2008), pp. 22–29. ISSN: 0740-7459. DOI: 10.1109/MS.2008.130.

[4] Christopher Bailey. *Pros and Cons of Type Hints*. Oct. 2019. URL: https://realpython.com/lessons/pros-and-cons-type-hints/ (visited on 01/31/2024).

[5] Luca Cardelli. “Type systems”. In: *ACM Computing Surveys (CSUR)* 28.1 (1996), pp. 263–264.

[6] Federico Cassano et al. “Type Prediction With Program Decomposition and Fill-in-the-Type Training”. In: *arXiv preprint arXiv:2305.17145* (May 2023).

[7] Kyle Daigle. *GitHub Octoverse 2023*. Nov. 2023. URL: https://github.blog/2023-11-08-the-state-of-open-source-and-ai/#the-most-popular-programming-languages (visited on 01/25/2024).

[8] Julien Delange. *What is Static Code Analysis and Why is it Important?* Dec. 2022. URL: https://www.codiga.io/blog/static-code-analysis-important/ (visited on 02/12/2024).

[9] Facebook. *Pyre: Performant type-checking for Python*. 2017. URL: https://github.com/facebook/pyre-check (visited on 01/17/2024).

[10] Wenhao Fan et al. *Example code and corresponding API call graph*. 2019. URL: https://www.researchgate.net/figure/Example-code-and-corresponding-API-call-graph_fig2_336655989 (visited on 01/27/2024).

[11] Cédric Fraboulet. *Why using type hints in large Python projects is a good idea?* Sept. 2021. URL: https://medium.com/@cedricfraboulet/why-using-type-hints-in-large-python-projects-is-a-good-idea-dd47cbdf8438 (visited on 01/31/2024).

[12] Zheng Gao, Christian Bird, and Earl T. Barr. “To Type or Not to Type: Quantifying Detectable Bugs in JavaScript”. In: IEEE, May 2017, pp. 758–769. ISBN: 978-1-5386-3868-2. DOI: 10.1109/ICSE.2017.75.

[13] Alexander S. Gillis. *What is static analysis (static code analysis)?* July 2020. URL: https://www.techtarget.com/whatis/definition/static-analysis-static-code-analysis (visited on 01/24/2024).

[14] Google. *pytype: A static type analyzer for Python code*. 2014. URL: https://github.com/google/pytype (visited on 01/17/2024).

[15] Luca Di Grazia and Michael Pradel. “The Evolution of Type Annotations in Python: An Empirical Study”. In: ACM, Nov. 2022, pp. 209–220. ISBN: 9781450394130. DOI: 10.1145/3540250.3549114.

[16] Stefan Hanenberg et al. “An empirical study on the impact of static typing on software maintainability”. In: *Empirical Software Engineering* 19 (5 Oct. 2014), pp. 1335–1382. ISSN: 1382-3256. DOI: 10.1007/s10664-013-9289-1.

[17] Vincent J. Hellendoorn et al. “Deep learning type inference”. In: ACM, Oct. 2018, pp. 152–162. ISBN: 9781450355735. DOI: 10.1145/3236024.3236051.

[18] Kevin Jesse, Premkumar T. Devanbu, and Toufique Ahmed. “Learning Type Annotation: Is Big Data Enough?” In: ACM, 2021. ISBN: 9781450385626. DOI: 10.1145/3468264.

[19] Kevin Jesse, Premkumar T. Devanbu, and Anand Sawant. “Learning to Predict User-Defined Types”. In: *IEEE Transactions on Software Engineering* 49 (4 Apr. 2023), pp. 1508–1522. ISSN: 0098-5589. DOI: 10.1109/TSE.2022.3178945.

[20] Jukka Lehtosalo. *mypy - Optional Static Typing for Python*. 2015. URL: https://mypy-lang.org (visited on 01/10/2024).

[21] Matt Makai. *Companies using Python*. 2018. URL: https://www.fullstackpython.com/companies-using-python.html (visited on 02/24/2024).

[22] Rabee Sohail Malik, Jibesh Patra, and Michael Pradel. “NL2Type: Inferring JavaScript Function Types from Natural Language Information”. In: IEEE, May 2019, pp. 304–315. ISBN: 978-1-7281-0869-8. DOI: 10.1109/ICSE.2019.00045.

[23] Microsoft. *Language Server Protocol*. 2016. URL: https://microsoft.github.io/language-server-protocol/ (visited on 06/25/2023).

[24] Microsoft. *Pyright: Static Type Checker for Python*. 2019. URL: https://github.com/microsoft/pyright (visited on 05/04/2023).

[25] Miladev95. *Static type vs Dynamic type*. Aug. 2023. URL: https://medium.com/@miladev95/static-type-vs-dynamic-type-2026d7810b7e (visited on 02/02/2024).

[26] Amir M. Mir et al. “Type4Py: Practical Deep Similarity Learning-Based Type Inference for Python”. In: ACM, May 2022, pp. 2241–2252. ISBN: 9781450392211. DOI: 10.1145/3510003.3510124.

[27] Stack Overflow. *Stack Overflow Developer Survey 2023*. 2023. URL: https://survey.stackoverflow.co/2023/#technology-most-popular-technologies (visited on 01/25/2024).

[28] Irene Vlassi Pandi et al. “OptTyper: Probabilistic Type Inference by Optimising Logical and Natural Constraints”. In: *arXiv preprint arXiv:2004.00348* (Apr. 2020).

[29] Yun Peng et al. “Static Inference Meets Deep Learning: A Hybrid Type Inference Approach for Python”. In: ACM, May 2022, pp. 2019–2030. ISBN: 9781450392211. DOI: 10.1145/3510003.3510038.

[30] Benjamin C Pierce. *Types and programming languages*. MIT press, 2002. ISBN: 0-262-16209-1.

[31] Michael Pradel et al. “TypeWriter: neural type prediction with search-based validation”. In: ACM, Nov. 2020, pp. 209–220. ISBN: 9781450370431. DOI: 10.1145/3368089.3409715.

[32] Python. *Typeshed - Collection of library stubs for Python, with static types*. Mar. 2015. URL: https://github.com/python/typeshed (visited on 01/17/2024).

[33] Baishakhi Ray et al. “A large scale study of programming languages and code quality in github”. In: ACM, Nov. 2014, pp. 155–165. ISBN: 9781450330565. DOI: 10.1145/2635868.2635922.

[34] Veselin Raychev, Martin Vechev, and Andreas Krause. “Predicting Program Properties from ”Big Code””. In: ACM, Jan. 2015, pp. 111–124. ISBN: 9781450333009. DOI: 10.1145/2676726.2677009.

[35] Guido van Rossum, Jukka Lehtosalo, and Łukasz Langa. *PEP 484 – Type Hints*. Sept. 2014. URL: https://peps.python.org/pep-0484/ (visited on 05/04/2023).

[36] Sandipan Roy. *Static Code Analysis for C++: Improving Code Quality and Security*. Mar. 2023. URL: https://blog.bytehackr.in/static-code-analysis-for-cpp#heading-what-is-static-code-analysis (visited on 02/12/2024).

[37] B.G. Ryder. “Constructing the Call Graph of a Program”. In: *IEEE Transactions on Software Engineering* SE-5 (3 May 1979), pp. 216–226. ISSN: 0098-5589. DOI: 10.1109/TSE.1979.234183.

[38] Vitalis Salis et al. “PyCG: Practical Call Graph Generation in Python”. In: IEEE, May 2021, pp. 1646–1657. ISBN: 978-1-6654-0296-5. DOI: 10.1109/ICSE43902.2021.00146.

[39] Lukas Seidel et al. “Learning Type Inference for Enhanced Dataflow Analysis”. In: *arXiv preprint arXiv:2310.00673* (Oct. 2023).

[40] Stav Shamir. *The other (great) benefit of Python type annotations*. May 2018. URL: https://medium.com/@shamir.stav_83310/the-other-great-benefit-of-python-type-annotations-896c7d077c6b (visited on 02/07/2024).

[41] Jeremy G. Siek and Walid Taha. *Gradual Typing for Functional Languages*. 2006. URL: http://schemeworkshop.org/2006/13-siek.pdf (visited on 02/01/2024).

[42] Devarshi Singh et al. “Evaluating How Static Analysis Tools Can Reduce Code Review Effort”. In: IEEE, Oct. 2017, pp. 101–105. ISBN: 978-1-5386-0443-4. DOI: 10.1109/VLHCC.2017.8103456.

[43] Daniel García Solla. *What is a Call Graph? And How to Generate them Automatically*. Jan. 2023. URL: https://www.freecodecamp.org/news/how-to-automate-call-graph-creation (visited on 06/19/2023).

[44] Eelco Visser. *Type Checking and Type Constraints*. URL: https://tudelft-cs4200-2020.github.io/lectures/2020/09/17/lecture4/ (visited on 02/22/2024).

[45] David Walker. *Type Checking*. URL: https://www.cs.princeton.edu/courses/archive/fall18/cos326/lec/17-type-checking.pdf (visited on 02/22/2024).

[46] Jiayi Wei, Greg Durrett, and Isil Dillig. “TypeT5: Seq2seq Type Inference using Static Analysis”. In: *arXiv preprint arXiv:2303.09564* (Mar. 2023).

[47] Jiayi Wei et al. “LambdaNet: Probabilistic Type Inference using Graph Neural Networks”. In: *arXiv preprint arXiv:2005.02161* (Apr. 2020).

[48] Wikipedia. *Type Inference*. URL: https://en.wikipedia.org/wiki/Type_inference (visited on 05/20/2023).

[49] Wikipedia. *Type System*. URL: https://en.wikipedia.org/wiki/Type_system#DYNAMIC (visited on 02/02/2024).

[50] Alastair Wilkes. *Static Code Analysis Approaches for Handling Code Quality*. Feb. 2023. URL: https://www.launchableinc.com/blog/static-code-analysis-approaches-for-handling-code-quality/ (visited on 02/12/2024).

[51] Zhaogui Xu et al. “Python probabilistic type inference with natural language support”. In: ACM, Nov. 2016, pp. 607–618. ISBN: 9781450342186. DOI: 10.1145/2950290.2950343.

[52] Yixuan Yan et al. “Scalable Demand-Driven Call Graph Generation for Python”. In: *arXiv preprint arXiv:2305.05949* (May 2023).

A
Examples of Successful Backtracking

An example run of PyHintSearch (top-3) where backtracking succeeded. This is for the Redis project, specifically the “connection.py” file: (See lines 2-40)

```
1  Processing file: .\connection.py
2  0: (’PythonRespSerializer’, ’__init__’)-buffer_cutoff -> str
3  1: (’PythonRespSerializer’, ’__init__’)-encode -> str
4  2: (’AbstractConnection’, ’_construct_command_packer’)-packer -> Optional[List[Any]]
5  2: (’AbstractConnection’, ’_construct_command_packer’)-packer -> dict
6  2: (’AbstractConnection’, ’_construct_command_packer’)-packer -> Optional[str]
7  2: (’AbstractConnection’, ’_construct_command_packer’)-packer ->
8  1: (’PythonRespSerializer’, ’__init__’)-encode -> Optional[str]
9  2: (’AbstractConnection’, ’_construct_command_packer’)-packer -> Optional[List[Any]]
10 2: (’AbstractConnection’, ’_construct_command_packer’)-packer -> dict
11 2: (’AbstractConnection’, ’_construct_command_packer’)-packer -> Optional[str]
12 2: (’AbstractConnection’, ’_construct_command_packer’)-packer ->
13 1: (’PythonRespSerializer’, ’__init__’)-encode -> bool
14 2: (’AbstractConnection’, ’_construct_command_packer’)-packer -> Optional[List[Any]]
15 2: (’AbstractConnection’, ’_construct_command_packer’)-packer -> dict
16 2: (’AbstractConnection’, ’_construct_command_packer’)-packer -> Optional[str]
17 2: (’AbstractConnection’, ’_construct_command_packer’)-packer ->
18 1: (’PythonRespSerializer’, ’__init__’)-encode ->
19 2: (’AbstractConnection’, ’_construct_command_packer’)-packer -> Optional[List[Any]]
20 2: (’AbstractConnection’, ’_construct_command_packer’)-packer -> dict
21 2: (’AbstractConnection’, ’_construct_command_packer’)-packer -> Optional[str]
22 2: (’AbstractConnection’, ’_construct_command_packer’)-packer ->
23 0: (’PythonRespSerializer’, ’__init__’)-buffer_cutoff -> int
24 1: (’PythonRespSerializer’, ’__init__’)-encode -> str
25 2: (’AbstractConnection’, ’_construct_command_packer’)-packer -> Optional[List[Any]]
26 2: (’AbstractConnection’, ’_construct_command_packer’)-packer -> dict
27 2: (’AbstractConnection’, ’_construct_command_packer’)-packer -> Optional[str]
28 2: (’AbstractConnection’, ’_construct_command_packer’)-packer ->
29 1: (’PythonRespSerializer’, ’__init__’)-encode -> Optional[str]
30 2: (’AbstractConnection’, ’_construct_command_packer’)-packer -> Optional[List[Any]]
31 2: (’AbstractConnection’, ’_construct_command_packer’)-packer -> dict
32 2: (’AbstractConnection’, ’_construct_command_packer’)-packer -> Optional[str]
33 2: (’AbstractConnection’, ’_construct_command_packer’)-packer ->
34 1: (’PythonRespSerializer’, ’__init__’)-encode -> bool
35 2: (’AbstractConnection’, ’_construct_command_packer’)-packer -> Optional[List[Any]]
36 2: (’AbstractConnection’, ’_construct_command_packer’)-packer -> dict
37 2: (’AbstractConnection’, ’_construct_command_packer’)-packer -> Optional[str]
38 2: (’AbstractConnection’, ’_construct_command_packer’)-packer ->
39 1: (’PythonRespSerializer’, ’__init__’)-encode ->
40 2: (’AbstractConnection’, ’_construct_command_packer’)-packer -> Optional[List[Any]]
41 3: (’AbstractConnection’, ’_construct_command_packer’)-return -> dict
42 3: (’AbstractConnection’, ’_construct_command_packer’)-return -> Dict[str, str]
43 3: (’AbstractConnection’, ’_construct_command_packer’)-return -> str
44 3: (’AbstractConnection’, ’_construct_command_packer’)-return ->
45 4: (’AbstractConnection’, ’_register_connect_callback’)-callback -> Callable
```

```
46 5: ('AbstractConnection', '_register_connect_callback')-return -> None
47 6: ('AbstractConnection', '_deregister_connect_callback')-callback -> Callable
48 7: ('AbstractConnection', '_deregister_connect_callback')-return -> None
49 8: ('AbstractConnection', '_connect')-return -> None
50 9: ('AbstractConnection', '_host_error')-return -> None
51 10: ('AbstractConnection', '_error_message')-exception -> Exception
52 11: ('AbstractConnection', '_error_message')-return -> None
53 12: ('AbstractConnection', '_send_ping')-return -> None
54 13: ('AbstractConnection', '_ping_failed')-error -> BaseException
55 14: ('AbstractConnection', '_ping_failed')-return -> None
56 15: ('Connection', '_connect')-return -> str
57 15: ('Connection', '_connect')-return -> Tuple[bytes, int]
58 15: ('Connection', '_connect')-return -> int
59 15: ('Connection', '_connect')-return ->
60 16: ('Connection', '_host_error')-return -> str
61 17: ('Connection', '_error_message')-exception -> Exception
62 18: ('Connection', '_error_message')-return -> str
63 19: ('SSLConnection', '__init__')-ssl_ca_certs -> bool
64 19: ('SSLConnection', '__init__')-ssl_ca_certs -> str
65 19: ('SSLConnection', '__init__')-ssl_ca_certs -> Tuple[str, int]
66 20: ('SSLConnection', '__init__')-ssl_ca_data -> bool
67 20: ('SSLConnection', '__init__')-ssl_ca_data -> str
68 20: ('SSLConnection', '__init__')-ssl_ca_data -> Optional[str]
69 21: ('SSLConnection', '__init__')-ssl_ca_path -> bool
70 21: ('SSLConnection', '__init__')-ssl_ca_path -> Union[str, Iterable[str]]
71 22: ('SSLConnection', '__init__')-ssl_cert_reqs -> bool
72 23: ('SSLConnection', '__init__')-ssl_certfile -> bool
73 23: ('SSLConnection', '__init__')-ssl_certfile -> str
74 23: ('SSLConnection', '__init__')-ssl_certfile -> Optional[bool]
75 24: ('SSLConnection', '__init__')-ssl_keyfile -> bool
76 24: ('SSLConnection', '__init__')-ssl_keyfile ->
77 25: ('SSLConnection', '__init__')-ssl_ocsp_context -> bool
78 26: ('SSLConnection', '__init__')-ssl_ocsp_expected_cert -> str
79 27: ('SSLConnection', '__init__')-ssl_password -> str
80 27: ('SSLConnection', '__init__')-ssl_password -> Union[str, List[str]]
81 28: ('SSLConnection', '_connect')-return -> str
82 28: ('SSLConnection', '_connect')-return -> Optional[bool]
83 28: ('SSLConnection', '_connect')-return -> bool
84 28: ('SSLConnection', '_connect')-return ->
85 29: ('UnixDomainSocketConnection', '__init__')-socket_timeout -> int
86 30: ('UnixDomainSocketConnection', '_connect')-return -> str
87 30: ('UnixDomainSocketConnection', '_connect')-return -> Tuple[bytes, int]
88 30: ('UnixDomainSocketConnection', '_connect')-return -> int
89 30: ('UnixDomainSocketConnection', '_connect')-return ->
90 31: ('UnixDomainSocketConnection', '_host_error')-return -> str
91 32: ('UnixDomainSocketConnection', '_error_message')-exception -> Exception
92 33: ('UnixDomainSocketConnection', '_error_message')-return -> str
93 34: ('ConnectionPool', 'from_url')-cls -> Optional[str]
94 35: ('BlockingConnectionPool', 'get_connection')-command_name -> str
95 36: ('BlockingConnectionPool', 'release')-connection -> psycopg2.extensions.connection
96 Found a combination of type annotations!
```

And another run of PyHintSearch (top-5) where backtracking succeeded. This is for the Django project, specifically on the “db/backends/ddl_references.py” file: (See lines 19-57)

```
1  Processing file: db\backends\ddl_references.py
2  0: ('Reference', 'references_table')-table -> str
3  1: ('Reference', 'references_column')-column -> List[str]
4  2: ('Reference', 'references_column')-table -> List[str]
5  3: ('Reference', 'rename_table_references')-new_table -> str
6  4: ('Reference', 'rename_table_references')-old_table -> str
7  5: ('Reference', 'rename_column_references')-new_column -> str
8  6: ('Reference', 'rename_column_references')-old_column -> str
9  7: ('Reference', 'rename_column_references')-table -> str
10 8: ('Table', '__init__')-quote_name -> str
11 9: ('Table', '__init__')-table -> str
12 10: ('Table', 'references_table')-table -> str
13 11: ('Table', 'references_table')-return -> Dict[str, Sequence[str]]
14 11: ('Table', 'references_table')-return -> str
15 11: ('Table', 'references_table')-return -> Dict[str, Any]
```

```
11: ('Table', 'references_table')-return ->
12: ('Table', 'rename_table_references')-new_table -> str
13: ('Table', 'rename_table_references')-old_table ->
14: ('TableColumns', '__init__')-columns -> int
15: ('TableColumns', '__init__')-table -> str
16: ('TableColumns', 'references_column')-column -> dict
16: ('TableColumns', 'references_column')-column -> List[Dict[str, Any]]
16: ('TableColumns', 'references_column')-column -> Dict[str, bool]
16: ('TableColumns', 'references_column')-column -> List[str]
16: ('TableColumns', 'references_column')-column -> tuple
16: ('TableColumns', 'references_column')-column ->
15: ('TableColumns', '__init__')-table -> List[List[str]]
16: ('TableColumns', 'references_column')-column -> dict
16: ('TableColumns', 'references_column')-column -> List[Dict[str, Any]]
16: ('TableColumns', 'references_column')-column -> Dict[str, bool]
16: ('TableColumns', 'references_column')-column -> List[str]
16: ('TableColumns', 'references_column')-column -> tuple
16: ('TableColumns', 'references_column')-column ->
15: ('TableColumns', '__init__')-table -> Optional[str]
16: ('TableColumns', 'references_column')-column -> dict
16: ('TableColumns', 'references_column')-column -> List[Dict[str, Any]]
16: ('TableColumns', 'references_column')-column -> Dict[str, bool]
16: ('TableColumns', 'references_column')-column -> List[str]
16: ('TableColumns', 'references_column')-column -> tuple
16: ('TableColumns', 'references_column')-column ->
15: ('TableColumns', '__init__')-table -> Optional[List[str]]
16: ('TableColumns', 'references_column')-column -> dict
16: ('TableColumns', 'references_column')-column -> List[Dict[str, Any]]
16: ('TableColumns', 'references_column')-column -> Dict[str, bool]
16: ('TableColumns', 'references_column')-column -> List[str]
16: ('TableColumns', 'references_column')-column -> tuple
16: ('TableColumns', 'references_column')-column ->
15: ('TableColumns', '__init__')-table ->
16: ('TableColumns', 'references_column')-column -> dict
16: ('TableColumns', 'references_column')-column -> List[Dict[str, Any]]
16: ('TableColumns', 'references_column')-column -> Dict[str, bool]
16: ('TableColumns', 'references_column')-column -> List[str]
16: ('TableColumns', 'references_column')-column -> tuple
16: ('TableColumns', 'references_column')-column ->
14: ('TableColumns', '__init__')-columns -> List[str]
15: ('TableColumns', '__init__')-table -> str
16: ('TableColumns', 'references_column')-column -> dict
17: ('TableColumns', 'references_column')-table -> dict
18: ('TableColumns', 'rename_column_references')-new_column -> str
19: ('TableColumns', 'rename_column_references')-old_column -> str
20: ('TableColumns', 'rename_column_references')-table -> str
21: ('Columns', '__init__')-col_suffixes -> str
22: ('Columns', '__init__')-columns -> str
22: ('Columns', '__init__')-columns ->
23: ('Columns', '__init__')-quote_name -> str
24: ('Columns', '__init__')-table -> str
25: ('IndexName', '__init__')-columns -> str
25: ('IndexName', '__init__')-columns -> Dict[str, Any]
25: ('IndexName', '__init__')-columns ->
26: ('IndexName', '__init__')-create_index_name -> str
27: ('IndexName', '__init__')-suffix -> str
28: ('IndexName', '__init__')-table -> str
29: ('IndexColumns', '__init__')-col_suffixes -> str
30: ('IndexColumns', '__init__')-columns -> str
31: ('IndexColumns', '__init__')-opclasses -> str
32: ('IndexColumns', '__init__')-quote_name -> str
33: ('IndexColumns', '__init__')-table -> str
34: ('ForeignKeyName', '__init__')-create_fk_name -> str
35: ('ForeignKeyName', '__init__')-from_columns -> Dict[str, Any]
36: ('ForeignKeyName', '__init__')-from_table -> Dict[str, Any]
37: ('ForeignKeyName', '__init__')-suffix_template -> str
38: ('ForeignKeyName', '__init__')-to_columns -> str
39: ('ForeignKeyName', '__init__')-to_table -> str
40: ('ForeignKeyName', 'references_table')-table -> Dict[str, Dict[str, str]]
41: ('ForeignKeyName', 'references_table')-return -> str
41: ('ForeignKeyName', 'references_table')-return -> bool
```

```
42: ('ForeignKeyName', 'references_column')-column -> Optional[List[str]]
42: ('ForeignKeyName', 'references_column')-column -> Dict[str, str]
43: ('ForeignKeyName', 'references_column')-table -> Optional[List[str]]
43: ('ForeignKeyName', 'references_column')-table -> Dict[str, str]
44: ('ForeignKeyName', 'rename_table_references')-new_table -> str
45: ('ForeignKeyName', 'rename_table_references')-old_table -> str
46: ('ForeignKeyName', 'rename_column_references')-new_column -> str
47: ('ForeignKeyName', 'rename_column_references')-old_column -> str
48: ('ForeignKeyName', 'rename_column_references')-table -> str
49: ('Statement', '__init__')-template -> str
50: ('Statement', 'references_table')-table ->
51: ('Statement', 'references_column')-column -> List[str]
52: ('Statement', 'references_column')-table -> List[str]
53: ('Statement', 'rename_table_references')-new_table -> str
54: ('Statement', 'rename_table_references')-old_table -> str
55: ('Statement', 'rename_column_references')-new_column -> str
56: ('Statement', 'rename_column_references')-old_column -> str
57: ('Statement', 'rename_column_references')-table -> str
58: ('Expressions', '__init__')-compiler -> Optional[str]
59: ('Expressions', '__init__')-expressions -> str
60: ('Expressions', '__init__')-quote_value -> str
61: ('Expressions', '__init__')-table -> str
62: ('Expressions', 'rename_table_references')-new_table ->
63: ('Expressions', 'rename_table_references')-old_table -> Mapping[str, Any]
63: ('Expressions', 'rename_table_references')-old_table ->
64: ('Expressions', 'rename_column_references')-new_column -> List[str]
65: ('Expressions', 'rename_column_references')-old_column -> str
66: ('Expressions', 'rename_column_references')-table -> str
Found a combination of type annotations!
```